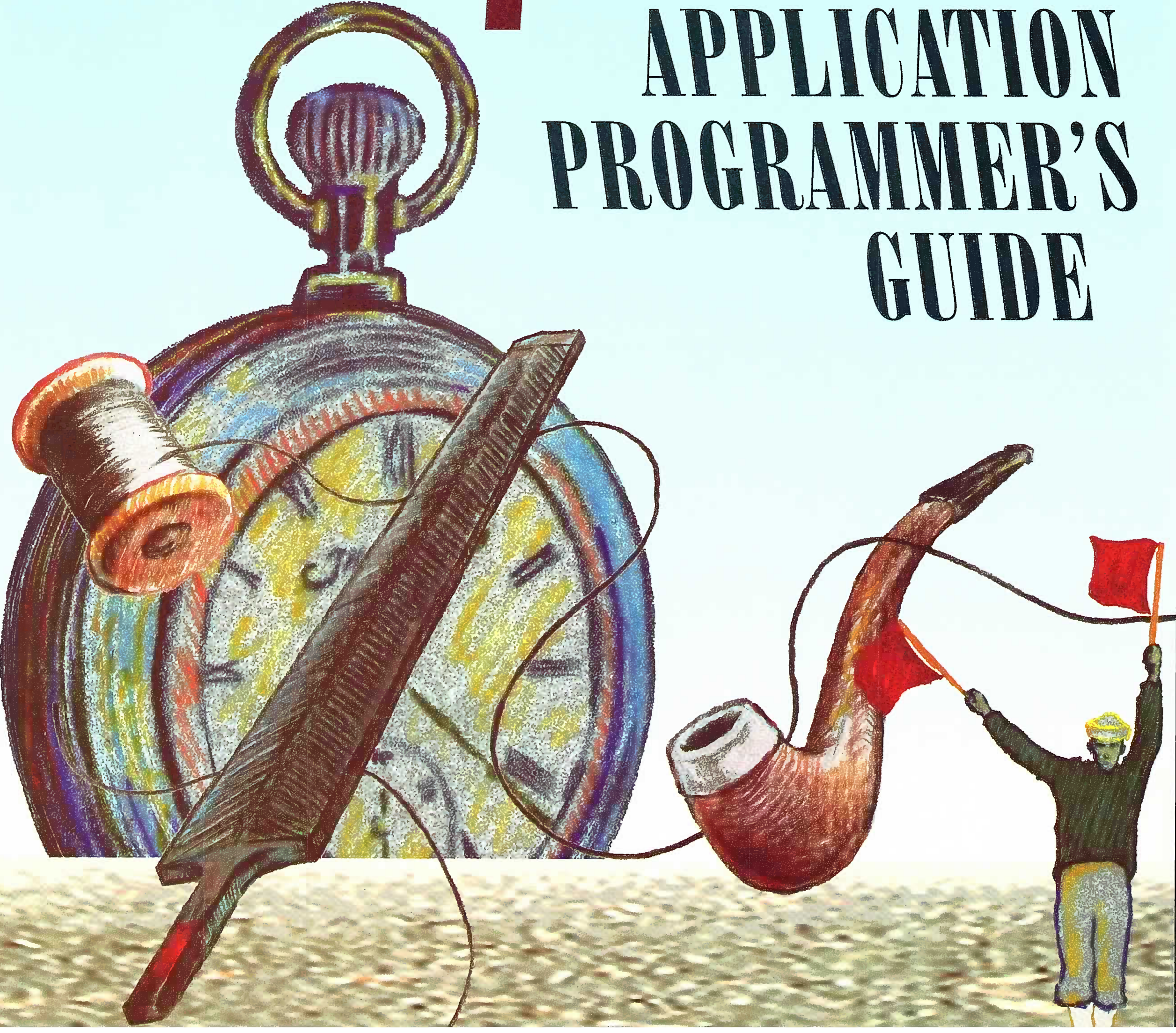
Jody Kelly • Craig Swearingen
Dawn Bezviner • Theodore Shrader

FOREWORD BY L.R. REISWIG, JR.

INCLUDES DISK
with ready-to-use code examples!

OS/2® 2.1 APPLICATION PROGRAMMER'S GUIDE



VNR Computer Library

OS/2® 2.1 APPLICATION PROGRAMMER'S GUIDE

OS/2® 2.1 APPLICATION PROGRAMMER'S GUIDE

JODY KELLY

CRAIG SWEARINGEN

DAWN BEZVINER

THEODORE SHRADER

VNR

VAN NOSTRAND REINHOLD
New York

Library of Congress Catalog Card Number 93-34418
ISBN 0-442-01736-7

Printed in the United States of America.

Van Nostrand Reinhold
115 Fifth Avenue
New York, NY 10003

International Thomson Publishing GmbH
Königswinterer Str. 418
53227 Bonn
Germany

International Thomson Publishing
Berkshire House,168-173
High Holborn, London WC1V 7AA
England

International Thomson Publishing Asia
221 Henderson Bldg. #05-1
Singapore 0315

Thomas Nelson Australia
102 Dodds Street
South Melbourne 3205
Victoria, Australia

International Thomson Publishing Japan
Kyowa Building, 3F
2-2-1 Hirakawacho
Chiyoda-ku, Tokyo 102
Japan

Nelson Canada
1120 Birchmount Road
Scarborough, Ontario
M1K 5G4, Canada

ARCFF 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2

Library of Congress Cataloging-in-Publication Data
OS/2 2.1 application programmer's guide / Jody Kelly [et al.].
p. cm.
Includes index.
ISBN 0-442-01736-7
1. Operating systems (Computers) 2. OS/2 (Computer file)
I. Kelly, Jody.
QA76.76.0630759 1993
005.4'469—dc20 93-34418
CIP

DEDICATIONS

- To my mother, Dr. Faye L. Kelly

  —J. K.

- To my parents, Bill and Betty Swearingen

  —C. S.

- To my parents, Celia and Morris Bezviner, and to my best friend, Julie Irvine

  —D. B.

- To my parents, Thomas and Michele Shrader

  —T. S.

CONTENTS

FOREWORD

OS/2 has helped to revolutionize the way people use workstations. This revolution started slowly with the 1.x versions of OS/2 and has mushroomed in capability and popularity. Both OS/2 2.0 and 2.1 can boast that they are integrating platforms that seamlessly preserve your DOS, Windows, and OS/2 1.x investments all on one desktop.

But as you begin to see what a 32-bit operating system can do, such as provide increased performance and virtual memory addressing, you will long for native 32-bit applications that can harness the full power and integrity of this 32-bit environment. This book will help you make this transition from 16-bit to 32-bit applications and help you write those programs that users are clamoring for.

This book gives an in-depth look at the suite of OS/2 2.1 Control Program APIs. Each topic, from memory management to queues, is covered in detail and includes extensive code examples. Many of the examples show the interaction between the APIs and how you can use them to accomplish tasks within your application. With this book, you'll give yourself many of the tools you'll need to build an OS/2 2.1 application.

Not only are the foundation topics covered, but this book also delves into how you can enable your application to work in the CID environment. CID stands for Configuration, Installation, and Distribution. It is a strategy that allows users and network administrators to easily install and configure your products across a LAN.

The LAN market has been growing and probably will continue to grow rapidly in the 1990s. Enabling your applications to work with CID will assure that your program product becomes a key part of the LAN environment.

This book also gives you an intermediate look at SOM, the System Object Model, complete with code examples that build upon each other. With your applications growing more and more complex every day, the importance of designing your code in an object-oriented way will increase your code reuse, reliability, and productivity. Use SOM to be your stepping stone to create more functionally rich applications.

This book also provides a brief introduction to the Distributed Computing Environment (DCE) and to the components of DCE that have been implemented on OS/2. Many developers are in the process of porting their OS/2 1.x applications to take advantage of the reliable, 32-bit, multitasking environment that both OS/2 2.0 and 2.1 provide.

This book will make the transition much easier by providing a detailed comparison between the OS/2 1.x to the OS/2 2.x Control Program APIs. Whether you are migrating your product from OS/2 1.x or DOS, discarding the segmented memory model in favor of OS/2 2.x's flat memory model will be just one of your many advantages.

By selecting this book, you have already made a decision to take advantage of a proven, robust environment. The authors bring many years of OS/2 programming and writing experience to this book. You'll find it a valuable tool as you reap the benefits of the 32-bit world that OS/2 2.1 provides.

Leland R. Reiswig, Jr.

PREFACE

"A journey of a thousand miles must begin with a single step."
—Lao–tzu

Welcome to OS/2 2.1 programming. If you're looking for help on writing applications to the OS/2 2.1 Control Program, you're in the right place. This book shows you how to use Control Program API calls, sometimes termed kernel calls, in your own applications. To use this book effectively, you should have some experience coding in a high-level language such as C and some understanding of either DOS or an earlier version of OS/2. Also, you should be interested in porting existing DOS or OS/2 1.x applications or in writing new applications for the OS/2 2.1 environment.

This book does not delve into programming for the Presentation Manager or the Workplace Shell. These very large topics deserve books of their own. This book is a programmer's guide, not an end user's guide. It does not explain how to install and configure OS/2 2.1, how to use the graphical user interface, nor how to install and run applications in the OS/2 2.1 operating system environment.

If you're a DOS programmer, you probably need to skim this preface and then focus on Chapters 1 and 2. When you're ready to write applications directly to OS/2 2.1, you can skim Chapter 3 and look closely at Chapters 4 through 15 for the APIs and code examples. Later on, you may want to look at the more advanced topics in last five chapters of the book. These chapters focus on how you can build your programs for the LAN and object-oriented environments.

If you're an OS/2 1.x 16-bit programmer, you can probably look briefly at Chapters 1 and 2 and skim Chapter 3. The new APIs and the code examples in Chapters 4 through 15 will be of immediate interest to you. The last five chapters can give you a taste of what's new.

If you're an OS/2 2.0 32-bit programmer, the most valuable sections of this book may well be the code examples in the chapters covering topics that you're interested in as well as the advanced topics in the last four chapters of the book. If you're coming from some entirely different operating system, the introductory information in Chapter 3 will probably be important to you, but you can safely skip Chapters 1 and 2.

Whatever your programming background, this book aims to provide useful, practical information that you can put into effect right away. Although not specifically a reference manual, this book contains enough detailed information on the APIs that you probably won't have to keep other books at your elbow as you write code. The parameters, flags, data structures, and code examples provided here should be enough to get you on your way. You can also use code from the diskette included with this book as examples for your own applications.

The rest of this book is organized as follows:

- ❏ Chapter 1 provides detailed differences between the 16-bit and the 32-bit APIs available in earlier releases of OS/2 and brings you up to date on the current APIs. It introduces topics that receive fuller treatment in later chapters, such as multitasking, and explains the OS/2 2.1 include files, thunking, and programming models.

- ❏ Chapter 2 shows you how to move from DOS to OS/2 2.1. It includes information on virtual DOS sessions and porting to OS/2 2.1.

- ❏ Chapter 3 introduces you to the OS/2 2.1 Control Program, or kernel, including both the architecture and the functionality. The chapter lists and briefly explains the 173 APIs provided in OS/2 2.1.

- ❏ Chapter 4 shows you how to compile C code to create executable files. It focuses on how you can create dynamic link libraries and explains the differences between static and dynamic linking.

- ❏ Chapter 5 introduces file management concepts and introduces the APIs for managing files.

- ❏ Chapter 6 provides information on file names, file systems, and file and device I/O.

- ❑ Chapter 7 shows you how to use extended attributes on files by using the relevant APIs and data structures.

- ❑ Chapter 8 gives details on what's new in OS/2 2.1 concerning memory management. Differences between private and shared memory are covered. You'll see how much easier your life will be after you move from the segmented to the flat memory model.

- ❑ Chapter 9 introduces the multitasking features of OS/2 2.1 such as sessions, processes, and threads.

- ❑ Chapter 10 gives details on the uses of semaphores and the types available to you. This chapter is one of three chapters (10, 11, and 12) that cover the Interprocess Communication (IPS) objects in OS/2 2.1.

- ❑ Chapter 11 explains the purposes and uses of pipes as well as the various types available to you.

- ❑ Chapter 12 explains how to use queues with semaphores, clients, and servers. It explains the APIs for creating, allocating, opening, reading, writing, purging, and closing queues.

- ❑ Chapter 13 explains how timers can be used in suspending threads, setting intervals, and managing asynchronous timers.

- ❑ Chapter 14 provides information on managing errors, exceptions, and messages. Return values, error classification, and exception handlers are among the topics covered.

- ❑ Chapter 15 describes how to use various tools to find and correct errors in your code.

- ❑ Chapter 16 introduces the fairly new programming concept known as Configuration-Installation-Distribution (CID). The chapter shows how to enable an application for installation and configuration across a local area network (LAN).

- ❑ Chapter 17 introduces object-oriented principles of programming, including encapsulation, inheritance, polymorphism, classes, methods, messages, and class libraries.

- ❑ Chapter 18 introduces the System Object Model (SOM), by which OS/2 2.1 implements object-oriented programming. It contains information on designing class hierarchies, subclassing, using the runtime environment, overriding methods, writing and compiling .CSC files, and developing libraries.

- ❑ Chapter 19 provides details on some of the advanced topics of programming with SOM. It contains information on bindings, customization, compiling, and language independence.

- ❑ Chapter 20 introduces the Distributed Computing Environment (DCE) as implemented for OS/2.

When you've mastered the information in this book, you should have a pretty good idea of what it takes to write applications to the OS/2 2.1 kernel. You'll also have three kinds of knowledge:

- ❑ Enough conceptual information to help you understand the purposes and uses of the various Control Program APIs as well as the other topics covered in this book.

- ❑ Enough reference information on the APIs to use them successfully in your applications.

- ❑ Enough code examples to use as models for your own code.

Use these examples as building blocks for your own applications.

ACKNOWLEDGMENTS

- Our IBM managers, Irene Hernandez, Greg Meyer, Chuck Schoepflin, and Robb Selby
- Chris Grisonich and Dianne Littwin of Van Nostrand Reinhold
- Wayne Lavitt for help with Ami Pro
- Dave Sorensen for comments on Chapter 2
- Paul Koenen and Gregory A. Fox for the code samples in Chapter 7 and Mark Molnar for comments on an early draft of the chapter
- Brad Swearingen for assistance in writing Chapter 15
- Tom Edel for comments on Chapter 16
- Gary Bourque for assistance in writing Chapter 19
- Ivan Milman for comments on an early draft of Chapter 20

TRADEMARKS

CHAPTER 1

Comparison of 16-Bit and 32-Bit APIs

"Shall I compare thee to a summer's day?" *– Shakespeare*

INTRODUCTION

To program to the OS/2 development model is to use the functions defined by the operating system. These functions are called Application Programming Interfaces (APIs). Generically speaking, OS/2 provides APIs to interface with hardware devices, such as a floppy disk or the screen, and interfaces to aid in the development of programs that access these devices.

The chapter introduces the programming model used by applications written to exploit the power of OS/2. This includes a comparison of the differences between the 16-bit and 32-bit OS/2 Control Program APIs.

OS/2 PROGRAMMING MODELS

OS/2 2.1 supports programs that have been built to run on DOS, DOS-Windows 3.1, OS/2 1.x, and OS/2 2.x. Programs written specifically for OS/2 2.x are able to exploit the 80386 architecture to its fullest. This can give your OS/2 32-bit application superior performance over the other application types.

Programs originally written for OS/2 1.x have the easiest migration path in porting to 32-bit OS/2 2.1. The OS/2 APIs have varied somewhat between the OS/2 1.x and the OS/2 2.1 versions, but they are quite similar with just a few exceptions. OS/2 2.1 offers full 16-bit, full 32-bit, or mixed 16-bit and 32-bit, programming models. Since programs written to OS/2 1.x, which are 16-bit applications, will run without change on OS/2 2.1, there exists the capability to write 16-bit applications.

These applications are limited to the capabilities of the 80286 architecture. Better yet, you can write programs for the 32-bit environment to exploit the capabilities of the 80386 architecture. To help ease the migration from a 16-bit program to 32-bit program, OS/2 2.1 allows a program to consist of a mixture containing both 16-bit and 32-bit code. The models consist of the following:

- ❑ A C compiler which produces 16-bit object code or one which produces 32-bit object code. Both compilers will be required for mixed 16-bit and 32-bit applications where individual modules are 16-bit or 32-bit enabled.
- ❑ The OS/2 1.3 Toolkit header files which describe the 16-bit APIs and the OS/2 2.1 Toolkit header files which describe the 32-bit APIs. Both toolkits will be required for mixed 16-bit and 32-bit applications where individual modules are 16-bit or 32-bit enabled.
- ❑ The OS/2 1.3 Toolkit LIB file, OS2.LIB, or the OS/2 2.1 Toolkit LIB files OS2286.LIB or OS2386.LIB. Mixed 16-bit and 32-bit applications must use OS2286.LIB or OS2386.LIB.
- ❑ The 16-bit LINK.EXE or 32-bit LINK386.EXE object code linking tool. Mixed and full 32-bit applications should use the LINK386.EXE tool.

OS2286.LIB and OS2386.LIB

The OS2286.LIB and OS2386.LIB files contain the link references to both the 16-bit and 32-bit OS/2 APIs. The OS2286.LIB file contains links to the 16-bit OS/2 APIs, for example, DosCreateQueue. The OS2286.LIB also contains links to the 32-bit APIs through an extended prefix that includes **32**, for example, Dos32CreateQueue. The OS2386.LIB contains links to the 32-bit APIs through its documented name and through the name with the **32** prefix, for example, the 32-bit DosCreateQueue or the Dos32CreateQueue. The OS2386.LIB file also contains links to the 16-bit APIs through an extended prefix that includes **16**, for example, Dos16CreateQueue.

Which LIB file you will want to use depends on your programming model. The 16-bit applications should use OS2286.LIB since the link references match the 16-bit header prototypes. The same argument is true for 32-bit applications using OS2386.LIB. For mixed 16-bit and 32-bit applications, it depends on whether most of your code is 16-bit or 32-bit. The reason is that you will have to provide the prototypes, definitions, and structures for one type of code, while the library will provide these items for the other type.

Include File Hierarchy

When a code module needs to use an OS/2 API, it is important to provide an API prototype before the API is used within the module. This is necessary to ensure that the parameters and the calling convention are correctly identified to the compiler. The OS/2 Toolkit provides C header files which contain the prototypes, definitions, and structures that the OS/2 APIs require. The primary header file is the file named os2.h. This file is the root file that includes all other header files that comprise the OS/2 API. To include all information related to all APIs, the code in Figure 1-1 would be used.

```
#define INCL_BASE   /* Include base OS API information   */
#define INCL_PM     /* Include Presentation Manager info */
#include <os2.h>
```

Figure 1-1. *Code to include all OS/2 include files.*

The hierarchy of files in Table 1-1 will be included when coded this way.

OS/2 1.3 INCLUDE FILES	OS/2 2.1 INCLUDE FILES
os2.h	os2.h
os2def.h	os2def.h
bse.h	bse.h
bsedos.h	bsedos.h
	bsetib.h
	bsememf.h
	bsexcpt.h
	bdcalls.h
bsesub.h	bsesub.h
bseerr.h	bseerr.h
	bsedev.h
	bseord.h

OS/2 1.3 INCLUDE FILES	OS/2 2.1 INCLUDE FILES
pm.h	pm.h
pmwin.h	pmwin.h
pmtypes.h	pmtypes.h
pmmle.h	pmmle.h
pmshl.h	pmshl.h
pmerr.h	pmerr.h
pmsei.h	pmsei.h
pmhelp.h	pmhelp.h
	os2nls.h
pmgpi.h	pmgpi.h
	pmbitmap.h
pmerr.h	pmerr.h
pmdev.h	pmdev.h
	pmwp.h
	pmerr.h
pmavio.h	pmavio.h
pmspl.h	pmspl.h
pmerr.h	pmerr.h
pmpic.h	pmpic.h
pmord.h	pmord.h
pmbitmap.h	
pmfont.h	pmfont.h
pmstddlg.	psmtddlg.h

Table 1-1. *OS/2 include file structure.*

The OS/2 API information is logically separated into the multiple files listed in Table 1-1 and, in some cases, further separated within each file. As you can see from Table 1-1, this is a large number of files to load into the compiler. If a particular module uses only a few APIs, you would waste compile time if you had to include all of these definitions.

The OS/2 header files provide a number of additional INCL_ flags that can be defined instead of INCL_BASE and INCL_PM. These allow a more precise specification for the information that should be included. The additional INCL_ flags that are supported within a header file are defined at the top of that header file.

The OS/2 2.1 include files contain definitions for only the 32-bit APIs accessed through the documented name. The exception is for the MOU, KBD, and VIO subsystems which are 16-bit only OS/2 subsystems. Applications that reference other 16-bit OS/2 APIs in 32-bit modules must provide the prototypes, definitions, and structures that they require.

MIXING 16-BIT AND 32-BIT CODE

Mixing 16-bit and 32-bit code in a program adds an additional layer of complexity to worry about. This mixing, also known as *thunking*, occurs when either 16-bit code needs to call a 32-bit function or 32-bit code needs to call a 16-bit function.

The 32-bit compiler offers you the ability to easily call 16-bit functions. The 32-bit compiler introduces the _Seg16, the _Far16 _Pascal, the #pragma stack16, and the /Gt+ compiler options to accomplish this. The _Seg16 data qualifier option is used to specify that a pointer should be a 16-bit pointer. The _Far16 _Pascal keywords are used to specify that a function uses the 16-bit calling convention.

The #pragma stack16 preprocessor statement is used to specify the size of a stack that is required for the 16-bit call if it needs to be something other than 4096 bytes in size. The /Gt+ compiler option compiles the code such that the 64K boundary will never be crossed for statically allocated data. The example program in Figure 1-2 calls the 16-bit version of the DosWrite OS/2 API.

```
#include <os2.h>
#include <string.h>

USHORT _Far16 _Pascal Dos16Write(SHORT, PVOID,
                             USHORT, PUSHORT);
int main()
{
    char * _Seg16 string;
    USHORT cb;

    string = "Hello hello!\n";
    Dos16Write(0, string, strlen((char *)string), &cb);
 }
```

Figure 1-2. *Program illustrating 32-bit to 16-bit thunking.*

The program in Figure 1-2 represents the principle that should be applied to any 16-bit function, whether it is an OS/2 API or one from your program. Notice that it is necessary to use the _Seg16 attribute for the string variable but not for the cb variable. Since the cb variable is not declared as a pointer, it should not be declared with the _Seg16 attribute even though a pointer is being passed to it with the Dos16Write function call.

Since the Dos16Write function call is prototyped to be a 16-bit function using the _Far16 _Pascal keywords, any parameters passed that are pointers are assumed to be 16-bit pointers. Therefore, no specification of this is required. Notice also that the string variable is cast to a 32-bit pointer before calling the strlen function. It is necessary to convert the 16-bit pointer before sending it to a 32-bit function.

When passing a structure pointer to a 16-bit function, you have to put the _Seg16 attribute on any pointer structure fields. It is also necessary to make sure the structure fields are on the boundaries that are expected by the 16-bit application. This will depend on the packing conventions used by the 16-bit code. The #pragma pack preprocessor directive can be coded around the declaration of the structure. It will indicate the appropriate packing that will align the fields within the structure on the boundaries required by the 16-bit function.

Thunking from 16-bit code to 32-bit code is not as convenient. The reason is that the 16-bit compilers do not provide any facility similar to the 32-bit compiler. This thunk can be accomplished by declaring a 32-bit function with the 16-bit calling convention (using _Far16 _Pascal). The 32-bit code should pass a pointer to the 16-bit code through a function pointer from a 32-bit to a 16-bit thunked function. The 16-bit code would then use the function pointer to make a far call. Figure 1-3 illustrates an example of this method.

```
#define INCL_DOSPROCESS
#include <os2.h>
#include <stdio.h>

typedef VOID ( * _Far16 _Pascal PFNEXITL)(USHORT);
USHORT _Far16 _Pascal DOS16EXITLIST(
      USHORT fFnCode,
      PFNEXITL pfnFunction);

VOID _Far16 _Pascal ExitList(USHORT usCode)
{
      printf("Exit list processing\n");
      DOS16EXITLIST(EXLST_EXIT, ExitList);
}
```

```
INT main(void)

{
      printf("Installing exit list\n");
   DOS16EXITLIST(EXLST_ADD, ExitList);
      return (0);
}
```

Figure 1-3. *Program illustrating 16-bit to 32-bit thunking.*

The program in Figure 1-3 uses the 16-bit DosExitList function and passes a pointer to a 32-bit function that the exit list processing will call.

CONTROL PROGRAM API CHANGES

The OS/2 Control Program API prototypes, structures, and definitions reside in the file bsedos.h which is part of the OS/2 Toolkit. This file contains sections, delineated by C preprocessor statements, that offer the ability to include only the portions relevant to a particular module. To indicate to the C preprocessor that a portion of the file should be included, the module should define the symbol that will reference that section of bsedos.h. For instance, to include the information necessary to use the OS/2 queue APIs, add the code in Figure 1-4 before the APIs are referenced in the module.

```
#define INCL_DOSQUEUES
#include <os2.h>
```

Figure 1-4. *Example inclusion of Queue API information.*

Most of the Control Program APIs have had some slight change from release OS/2 1.3 to 2.1. The majority of these were only simple data type changes such as changing the data type of a parameter from USHORT to ULONG. Many APIs were also renamed to conform to consistent standards of naming. Some APIs were removed since they no longer apply in the 32-bit environment. There were a few major changes, most notably in the areas of memory management, semaphores, signals, and file functions.

The following tables of APIs are grouped according to the groupings found within bsedos.h. The tables show a mapping from the 16-bit API to an equivalent 32-bit API where one exists. With the exception of the memory management, semaphores, signals, and file function APIs, the equivalent API primarily presents a one-for-one mapping from the 16-bit version.

Thread and Process APIs

The thread and process OS/2 API prototypes and definitions are included by specifying INCL_DOSPROCESS and INCL_DOSINFOSEG. With the 32-bit release, the INCL_DOSINFOSEG specification is no longer needed. Table 1-2 shows the comparable 16-bit and 32-bit thread and process APIs. The codes in the center column are explained at the end of the table.

16-BIT API	CODES	32-BIT EQUIVALENT
DosBeep	DT	DosBeep
DosCreateThread	NF	DosCreateThread
DosCWait	N,DT	DosWaitChild
DosEnterCritSec	NC	DosEnterCritSec
DosExecPgm	DT	DosExecPgm
DosExit	DT	DosExit
DosExitCritSec	NC	DosExitCritSec
DosExitList	DT	DosExitList
DosGetEnv	NF	DosGetInfoBlocks
DosGetInfoSeg	NF	DosGetInfoBlocks, DosQuerySysInfo
DosGetPrty	NF	DosGetInfoBlocks
DosGetPID	NF	DosGetInfoBlocks
DosGetPPID	NF	DosGetInfoBlocks
DosKillProcess	DT	DosKillProcess
DosPTrace	NF	DosDebug
DosQSysInfo	NF	DosQuerySysInfo
DosResumeThread	NC	DosResumeThread
DosSetPrty	N,DT	DosSetPriority
DosSuspendThread	NC	DosSuspendThread

Codes:

DT	=	API parameter data type changes, such as changing from USHORT to ULONG.
N	=	API name change.
NC	=	No change.

```
NF = New function beyond API parameter data
     type changes, as discussed below.
```

Table 1-2. *16-bit and 32-bit thread and process APIs.*

The DosCreateThread API experienced several changes:

- ❑ The thread's stack is created for the caller. With the 16-bit version, it is necessary to issue a DosAllocSeg call and pass the stack segment to the DosCreateThread API.

- ❑ A parameter can be passed to the thread creation function. This is valuable for communicating information to the thread that can be used during its initialization.

- ❑ The thread can be created in the suspended mode. DosResumeThread can be used to wake the thread at an appropriate time.

The DosGetEnv, DosGetPrty, DosGetPID, and DosGetPPID APIs have all been removed, but the same information can be retrieved through the 32-bit DosGetInfoBlocks API.

The DosGetInfoBlocks API essentially replaces the DosGetInfoSeg 16-bit API, which also has been removed. Some of the information that used to be available through the DosGetInfoSeg is now available through the DosQuerySysInfo 32-bit API. Also, the DosQSysInfo 16-bit API is replaced by the DosQuerySysInfo API.

The DosPTrace API is used for writing debugger applications. This functionality has been completely rewritten with the DosDebug API.
Table 1-3 lists an API that has no equivalent mapping in the INCL_DOSPROCESS grouping.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
	DosWaitThread

Table 1-3. *Thread and process APIs with no equivalent.*

The DosWaitThread 32-bit API has no counterpart. The DosWaitThread allows a thread to wait for another thread within the same process to end.

File Management APIs

The file management OS/2 API prototypes and definitions are included by specifying INCL_DOSFILEMGR. The file management APIs had a significant number of name changes. There are also a few functions no longer supported. Table 1-4 shows the comparable 16-bit and 32-bit file management APIs. The codes in the center column are explained at the end of the table.

16-BIT APIs	CODES	32-BIT APIs
DosBufReset	N	DosResetBuffer
DosChDir	N,NR	DosSetCurrentDir
DosChFilePtr	N,DT	DosSetFilePtr
DosClose	NC	DosClose
DosCopy	NR	DosCopy
DosDelete	NR	DosDelete
DosDupHandle	NC	DosDupHandle
DosEditName	DT	DosEditName
DosEnumAttribute	NR	DosEnumAttribute
DosFSAttach	NR	DosFSAttach
DosFSCtl	NR	DosFSCtl
DosFileLocks	N,NF	DosSetFileLocks
DosFindClose	NC	DosFindClose
DosFindFirst, DosFindFirst2	NR,NF	DosFindFirst
DosFindNext	DT	DosFindNext
DosOpen, DosOpen2	DT	DosOpen
DosQCurDir	N,DT	DosQueryCurrentDir
DosQCurDisk	N,DT	DosQueryCurrentDisk
DosQFHandleState	N,DT	DosQueryFHState
DosQFSAttach	N,NR	DosQueryFSAttach
DosQFSInfo	N,DT	DosQueryFSInfo
DosQFileInfo	N,DT	DosQueryFileInfo
DosQFileMode	NF	DosQueryFileInfo
DosQHandType	N,DT	DosQueryHType

16-BIT APIs	CODES	32-BIT APIs
DosQPathInfo	N,NR	DosQueryPathInfo
DosQVerify	N,DT	DosQueryVerify
DosRead	DT	DosRead
DosRmDir	N,NR	DosDeleteDir
DosSelectDisk	N,DT	DosSetDefaultDisk
DosSetFHandState	N,DT	DosSetFHState
DosSetFSInfo	DT	DosSetFSInfo
DosSetFileInfo	DT	DosSetFileInfo
DosSetFileMode	NF	DosSetFileInfo
DosSetMaxFH	DT	DosSetMaxFH
DosSetPathInfo	NR	DosSetPathInfo
DosSetVerify	DT	DosSetVerify
DosShutdown	NC	DosShutdown
DosWrite	DT	DosWrite

Codes:

DT	=	API parameter data type changes such as changing from SHORT to ULONG.
N	=	API name change.
NC	=	No change.
NF	=	New function beyond API parameter data type changes, as discussed further below.
NR	=	Removed the reserved parameter.

Table 1-4. *16-bit and 32-bit file management APIs.*

The DosSetFileLocks API has two significant functions beyond what the 16-bit DosFileLocks API provides. It has the ability to specify a maximum timeout value that the process will wait for the requested locks. It also allows the option of specifying whether the locked file range is sharable between processes in a read-only state.

The DosFindFirst function has two significant changes beyond what the 16-bit DosFindFirst and DosFindFirst2 APIs provide. The function has been enhanced to allow the specification of exclusion flags. These flags can specify that a file must be a directory, system, archived, hidden, or read-only file. The 16-bit version allows only the specification of what attributes may be present.

This characteristic provides for a more appropriate specification for the search. The 32-bit DosFindFirst API also supports specifying which level (1, 2, or 3) of information to return. The 16-bit version always returned all 3 levels of information.

Table 1-5 shows the file management APIs that have no equivalent mappings in the INCL_DOSFILEMGR grouping.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosFileIO	
DosFindNotifyClose	
DosFindNotifyFirst	
DosFindNotifyNext	
DosReadAsync	
	DosCancelLockRequest
	DosForceDelete
	DosSetRelMaxFH
	DosProtectClose
	DosProtectEnumAttribute
	DosProtectOpen
	DosProtectQueryFHState
	DosProtectQueryFileInfo
	DosProtectRead
	DosProtectSetFHState
	DosProtectSetFileInfo
	DosProtectSetFileLocks
	DosProtectSetFilePtr
	DosProtectSetFileSize
	DosProtectWrite

Table 1-5. *File management APIs with no equivalent.*

The DosFileIO API is used to do file locking, to move the file pointer, and to do file I/O. There are also individual APIs available to do these operations.

The DosReadAsync and DosWriteAsync 16-bit APIs have no equivalent 32-bit versions. An asynchronous read or write can be handled on a separate thread using an event semaphore to notify about the completion of the asynchronous operation. The DosCancelLockRequest 32-bit API can be used to cancel a lock that was set by the DosSetFileLocks API. The DosForceDelete API is available to delete a file in such away that it can not be recovered with the UNDELETE command. Files deleted with DosDelete can still be recovered with the UNDELETE command.

The DosSetRelMaxFH 32-bit API is used to set the number of file handles for the current process. This API differs from the DosSetMaxFH API in that the operating system may delay or disregard a request to lower the number of file handles for the current process.

Memory Management APIs

The memory management OS/2 API prototypes and definitions are included by specifying INCL_DOSMEMMGR. The memory management APIs have been completely rewritten to take advantage of the 32-bit architecture. A request for memory returns a memory *object* instead of a *segment*. Memory objects are not restricted to 64K, which is the maximum segment size, but instead can have a much larger size up to 512 MB. There are also a number of functions no longer supported. Table 1-6 shows the comparable 16-bit and 32-bit equivalent memory management APIs. The codes in the center column are explained at the end of the table.

16-BIT API	CODE	32-BIT API
DosAllocSeg, DosAllocHuge, DosCreateCSAlias	N,NF	DosAllocMem
DosAllocShrSeg	NF	DosAllocSharedMem
DosGetShrSeg	N,NF	DosGetNamedSharedMem
DosGetSeg	N,NF	DosGetSharedMem
DosGiveSeg	N,NF	DosGiveSharedMem
DosFreeSeg	N,DT	DosFreeMem
DosSubAlloc	N,DT	DosSubAllocMem
DosSubFree	N,DT	DosSubFreeMem
DosSubSet	N,DT	DosSubSetMem

```
Codes:
   DT = API parameter data type changes such as changing
        from USHORT to ULONG.
   N  = API name change.
   NF = New function beyond API parameter data.
        type changes, as discussed further below.
```

Table 1-6. *16-bit and 32-bit memory management APIs.*

The DosAllocSeg, DosAllocHuge, and DosCreateCSAlias 16-bit APIs have all been replaced with the DosAllocMem 32-bit API. The DosAllocHuge API was used to create memory buffers that would span across the 64K boundary. This boundary no longer exists in 32-bit.

The DosAllocMem API offers a PAG_EXECUTE option, which creates a memory object where code can be written to and then executed. This is the function that the 16-bit DosCreateCSAlias API provided.

Like the DosAllocMem API, the DosAllocSharedMem API offers the option of specifying allocation attributes and desired access protection attributes. These attributes specify characteristics about both the memory object and the allocation of the memory object.

The DosGetNamedSharedMem, DosGetSharedMem, and DosGiveSharedMem also let you specify the access protection attributes you want to use.
Table 1-7 shows the memory management APIs that have no equivalent mappings in the INCL_DOSMEMMGR grouping.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosGetHugeShift	
DosMemAvail	
DosLockSeg	
DosReallocHuge	
DosReallocSeg	
DosSizeSeg	
DosUnlockSeg	
	DosSetMem
	DosSubUnsetMem

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
	DosQueryMem

Table 1-7. *Memory management APIs with no equivalent.*

Since huge segments no longer exist in the 32-bit programming model, the DosGetHugeShift API is no longer required.

The DosMemAvail API is used in the segmented memory model to help measure system-memory load and has no corresponding function in the 32-bit flat model version.

The DosLockSeg API and the DosUnlockSeg 16-bit API were used to lock and unlock discardable segments into memory. The 32-bit flat memory model does not support the concept of discardable pages.

DosReallocSeg and DosReallocHuge 16-bit APIs are used to change the allocation size of the segments that were previously allocated. These APIs are no longer needed because the 64K boundary condition has been lifted in the 32-bit flat memory model.

You also have the ability to create the memory object without committing all of the memory at creation time.

The DosSizeSeg 16-bit API is used to query the size of a segment and has no corresponding 32-bit API.
The DosSetMem 32-bit API is used to change the allocation attributes and desired protection attributes of a memory object after it has been created.

The DosSubUnsetMem 32-bit API must be called before a memory object is freed if it had previously been setas suballocatable by a call to the DosSubSetMem API.

The DosQueryMem 32-bit API can be used to query the allocation and protection attributes that the memory object currently has set.

Semaphore APIs

The semaphore OS/2 API prototypes and definitions are included by specifying INCL_DOSSEMAPHORES. The semaphore APIs have been completely rewritten for the 32-bit version. Both 16-bit and 32-bit semaphore APIs provide the ability for managing mutual exclusion, event, and multiple semaphores. The main difference is that the 32-bit APIs have the same usage model for all three categories of APIs. The usage model is the following:

```
<Create the semaphore> <Open the semaphore>

<Request or Release> <Post or Wait> on the semaphore

<Close the semaphore>
```

Table 1-8 shows the comparable semaphore APIs with an equivalent mapping in the INCL_DOSSEMAPHORES grouping. The code is explained at the end of the table.

16-BIT API	CODE	32-BIT API
DosFSRamSemClear	NF	DosReleaseMutexSem
DosFSRamSemRequest	NF	DosRequestMutexSem
DosCloseSem	NF	DosCloseEventSem, DosCloseMutexSem
DosCreateSem	NF	DosCreateEventSem, DosCreateMutexSem
DosMuxSemWait	NF	DosWaitMuxWaitSem
DosOpenSem	NF	DosOpenEventSem, DosOpenMutexSem
DosSemClear	NF	DosPostEventSem
DosSemRequest	NF	DosRequestMutexSem
DosSemSet	NF	DosResetEventSem
DosSemWait	NF	DosWaitEventSem

Code:
NF = New function beyond API parameter data type changes, as discussed below.

Table 1-8. *16-bit and 32-bit semaphore APIs.*

Mutual exclusion was accomplished by means of the DosFSRamSemClear and the DosFSRamSemRequest 16-bit APIs. The Dos*MutexSem 32-bit APIs are the corresponding APIs to accomplish mutual exclusion. These semaphores must be created before a request or release of the semaphore can be requested and should be closed when no longer needed. The fast-safe-RAM semaphores are available only within the current process, whereas the 32-bit mutual exclusion semaphores can be shared.

Mutual exclusion is also supported through the DosCreateSem, DosSemRequest, DosSemClear, and DosCloseSem 16-bit APIs.

Corresponding 32-bit APIs exist in OS/2 2.1. They include DosCreateMutexSem, DosquestMutexSem, DosReleaseMutexSem, and DosCloseMutexSem.

Signaling events is accomplished by means of the DosCreateSem, DosSemClear, DosSemWait, and DosCloseSem 16-bit APIs.

The corresponding 32-bit APIs exist in OS/2 2.1. They include DosCreateEventSem, DosPostEventSem, DosWaitEventSem, and DosCloseEventSem.

Multiple semaphore waiting is accomplished through the 16-bit DosMuxSemWait API. The 32-bit model for waiting on multiple semaphores starts with the creation of a muxwait semaphore through the DosCreateMuxWaitSem API. Once created, the DosWaitMuxWaitSem API can be used to wait for one or all of the semaphores to get posted or released. When it is no longer needed, close the muxwait semaphore by means of the DosCloseMuxWaitSem API.

Table 1-9 lists the semaphore APIs that have no equivalent mappings.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosSemSetWait	
	DosAddMuxWaitSem
	DosDeleteMuxWaitSem
	DosCloseMuxWaitSem
	DosCreateMuxWaitSem
	DosOpenMuxWaitSem
	DosQueryEventSem
	DosQueryMutexSem
	DosQueryMuxWaitSem

Table 1-9. *Semaphore APIs with no equivalent.*

Given the 32-bit semaphore model, there are additional APIs that are both required and helpful to managing this model. This includes APIs to query significant information about a semaphore as well as to tailor muxwait semaphores.

Date and Time APIs

The date and time OS/2 API prototypes and definitions are included by specifying INCL_DOSDATETIME.

These APIs are used for querying and setting the current date or time, suspending a thread for a specific time interval, and operating an asynchronous timer. There have been only name changes within this group of APIs.

Table 1-10 shows the comparable date and time APIs in the INCL_DOSDATETIME grouping. The codes are explained at the end of the table.

16-BIT API	CODES	32-BIT API
DosGetDateTime	NC	DosGetDateTime
DosSetDateTime	NC	DosSetDateTime
DosSleep	NC	DosSleep
DosTimerAsync	N	DosAsyncTimer
DosTimerStart	N	DosStartTimer
DosTimerStop	N	DosStopTimer

```
Codes:
  N  = API name change.
  NC = No change.
```

Table 1-10. *16-bit and 32-bit date and time APIs.*

Resource and Dynamic Link Library APIs

The resource and dynamic link library (DLL) OS/2 API prototypes and definitions are included by specifying INCL_DOSRESOURCES and INCL_DOSMODULEMGR. Program resources such as icons, menus, and accelerators can be bound either to an executable file (EXE) or to a DLL. The DosGetResource API is used to gain access to a particular resource.

Either code or resources can be stored in a DLL. Code and resources stored in a DLL can be dynamically accessed by first loading the DLL with the DosLoadModule API.

With the exception of two new 32-bit APIs within this grouping, these APIs have remained relatively unchanged from the 16-bit version.

Table 1-11 shows the comparable 16-bit and 32-bit resource and DLL APIs in the INCL_DOSRESOURCES and INCL_DOSMODULEMGR groupings.

16-BIT API	CODES	32-BIT API
DosFreeModule	NC	DosFreeModule
DosFreeResource	NC	DosFreeResource
DosGetModHandle	N	DosQueryModuleHandle
DosGetModName	N,DT	DosQueryModuleName
DosGetProcAddr	N,NF	DosQueryProcAddr
DosGetResource, DosGetResource2	N,DT	DosGetResource
DosLoadModule	N,DT	DosLoadModule

```
Codes:
   DT = API parameter  data  type  changes such
        as changing from USHORT to ULONG.
   N  = API name change.
   NC = No change.
   NF = New function beyond  API  parameter
        data type changes, as discussed below.
```

Table 1-11. *16-bit and 32-bit resource and DLL APIs.*

The DosGetProcAddr 16-bit API is used to query the address of a function, by function name, that resides in a DLL. The DosQueryProcAddr 32-bit version also offers the option of querying a function by its ordinal position.

The ordinal position of a function within a DLL is specified in the module definitions file (.DEF) that is referenced during the LINK386 step while building the program.

Table 1-12 lists the resource and DLL APIs that have no equivalent mappings.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
	DosQueryResourceSize
	DosQueryProcType

Table 1-12. Resource and DLL APIs with no equivalent.

The DosQueryResourceSize API provides the ability to query the size in bytes that a resource requires.

The DosQueryProcType API provides the ability to determine if a procedure within a DLL is written in 16-bit or 32-bit code.

Code-Page Management APIs

The code-page management OS/2 API prototypes and definitions are included by specifying INCL_DOSNLS. These APIs are used for querying and setting information related to the code page that the process is using.

With the exception of API name changes, these APIs have had only minor data type changes made to the 32-bit versions.

Table 1-13 shows the comparable 16-bit and 32-bit code-page management APIs in the INCL_DOSNLS grouping. The codes in the center column are explained at the end of the table.

16-BIT API	CODES	32-BIT API
DosCaseMap	N,DT	DosMapCase
DosGetCollate	N,DT	DosQueryCollate
DosGetCP	N,DT	DosQueryCP
DosGetCtryInfo	N,DT	DosQueryCtryInfo
DosGetDBCSEv	N,DT	DosQueryDBCSEnv
DosSetProcCp	N,NR	DosSetProcessCp

```
Codes:
     DT = API Parameter data type changes such as
          changing from USHORT to ULONG.
     N  = API name change.
```

```
NR = Removed the reserved parameter.
```

Table 1-13. *16-bit and 32-bit code-page management APIs.*

Table 1-14 contains the only 16-bit code-page API that has no equivalent 32-bit API in this grouping.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosSetCp	

Table 1-14. *Code-page management API that has no equivalent.*

The DosSetProcessCp is the 32-bit API that is to be used to set the code-page for the current process. DosSetCp has no 32-bit equivalent.

Signal and Exception APIs

The 16-bit signal management OS/2 API prototypes and definitions are included by specifying INCL_DOSSIGNALS. Signaling capabilities have been removed from the 32-bit version and have been integrated into the exception management APIs.

The 32-bit exception management OS/2 API prototypes and definitions are included by specifying INCL_DOSEXCEPTIONS.

The signal APIs are used to manage the CTRL+C, the CTRL+BREAK, and the KILLPROCESS events that could happen on a full-screen or windowable OS/2 application.

The exception APIs allow you to manage these same signals and also the general protection exceptions in the application. In some cases this might allow the application to terminate properly or recover from the exception condition.

Table 1-15 shows the 16-bit and the 32-bit exception management APIs that have no equivalent mappings.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosFlagProcess	
DosHoldSignal	

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosSendSignal	
DosSetSigHandler	
	DosAcknowledgeSignalException
	DosEnterMustComplete
	DosExitMustComplete
	DosRaiseException
	DosSendSignalException
	DosSetExceptionHandler
	DosSetSignalExceptionFocus
	DosUnsetExceptionHandler
	DosUnwindException

Table 1-15. *Exception management APIs with no equivalent.*

Miscellaneous APIs

The miscellaneous OS/2 API prototypes and definitions are included by specifying INCL_DOSMISC. These APIs are a miscellaneous grouping of APIs that provide varying capabilities. These functions have had few changes between the 16-bit and 32-bit versions.

Table 1-16 shows the comparable 16-bit and 32-bit miscellaneous APIs within the INCL_DOSMISC grouping.

16-BIT API	CODES	32-BIT API
DosErrClass	DT	DosErrClass
DosError	DT	DosError
DosGetEnv	NF	DosGetInfoBlocks
DosGetMessage	DT	DosGetMessage
DosGetVersion	NF	DosQuerySysInfo
DosInsMessage	N,DT	DosInsertMessage
DosPutMessage	DT	DosPutMessage
DosQSysInfo	NF	DosQuerySysInfo
DosScanEnv	NC	DosScanEnv

16-BIT API	CODES	32-BIT API
DosSearchPath	DT	DosSearchPath

Codes:

DT	=	API parameter data type changes such as changing from USHORT to ULONG.
N	=	API name change.
NC	=	No change.
NF	=	New function beyond API parameter data type changes, as discussed below.

Table 1-16. *16-bit and 32-bit miscellaneous APIs.*

The DosGetEnv and DosGetVersion 16-bit APIs have been removed, but you can retrieve the same information through the 32-bit DosGetInfoBlocks API and the DosQuerySysInfo API respectively.

Table 1-17 lists the miscellaneous APIs that have no equivalent mappings in the INCL_DOSMISC grouping.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosGetMachineMode	
DosSetVec	
	DosQueryMessageCp

Table 1-17. *Miscellaneous APIs with no equivalent.*

The DosGetMachineMode API is not needed in the 32-bit environment as this API is used to query whether the processor is running in the DOS mode or the OS/2 mode.

The DosSetVec 16-bit API has no 32-bit counterpart, but the function it provides can be accessed using the 32-bit exception management APIs. Several of the APIs in this group are used for managing messages.

One additional 32-bit API exists, DosQueryMessageCp, which is used for querying the code page the message was created with.

Monitor APIs

The 16-bit monitor OS/2 API prototypes and definitions are included by specifying INCL_DOSMONITORS. The monitor APIs are used to gain access to character stream devices, such as a keyboard, and monitor for events on these devices. The monitor functions have no equivalent 32-bit version in OS/2 2.1.

Table 1-18 lists the 16-bit monitor APIs that have no equivalent mappings.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosMonClose	
DosMonOpen	
DosMonRead	
DosMonReg	
DosMonWrite	

Table 1-18. *Monitor APIs with no equivalent.*

Queue Management APIs

The queue management OS/2 API prototypes and definitions are included by specifying INCL_DOSQUEUES. These APIs are used for managing a queue that can be used for interprocess communication. These functions have remained relatively unchanged between the 16-bit and 32-bit version.

Table 1-19 shows the comparable 16-bit and 32-bit queue APIs within the INCL_DOSQUEUES grouping. The codes are explained at the end of the table.

16-BIT API	CODES	32-BIT API
DosCloseQueue	NC	DosCloseQueue
DosCreateQueue	NF	DosCreateQueue
DosOpenQueue	DT	DosOpenQueue
DosPeekQueue	DT	DosPeekQueue
DosPurgeQueue	NC	DosPurgeQueue
DosQueryQueue	DT	DosQueryQueue
DosReadQueue	DT	DosReadQueue
DosWriteQueue	DT	DosWriteQueue

```
Codes:
     DT =  API parameter data type changes such as
           changing from USHORT to ULONG.
     NC =  No change.
     NF =  New function beyond API parameter data type
           changes, as discussed below.
```

Table 1-19. *16-bit and 32-bit queue APIs.*

The DosCreateQueue 32-bit version also offers the QUE_CONVERT_ADDRESS option, which automatically converts the data addresses written to the queue by 16-bit processes to 32-bit addresses.

Session Management APIs

The session management OS/2 API prototypes and definitions are included by specifying INCL_DOSSESMGR. These APIs are used for managing a session that is created by the application.

These functions have remained relatively unchanged between the 16-bit and 32-bit version.

Table 1-20 shows the comparable 16-bit and 32-bit session management APIs in the INCL_DOSSESMGR grouping.

16-BIT API	CODES	32-BIT API
DosStartSession	DT	DosStartSession
DosSetSession	DT	DosSetSession
DosSelectSession	DT,NR	DosSelectSession
DosStopSession	DT,NR	DosStopSession
DosQAppType	N,DT	DosQueryAppType

```
Codes:
     DT = API parameter data type changes such as
          changing from SHORT to ULONG.
     N  = API name change.
     NR = Removed the reserved parameter.
```

Table 1-20. *16-bit and 32-bit session management APIs.*

Table 1-21 lists the session management that has no equivalent mapping.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosSMRegisterDD	

Table 1-21. *Session management API with no equivalent.*

Device Management APIs

The device management OS/2 API prototypes and definitions are included by specifying INCL_DOSDEVICES. These APIs are used for managing low-level devices by the application. These functions have remained relatively unchanged between the 16-bit and 32-bit version.

Table 1-22 shows the APIs with an equivalent mapping API within the INCL_DOSDEVICES grouping.

16-BIT API	CODE	32-BIT API
DosDevConfig	DT	DosDevConfig
DosDevIOCtl, DosDevIOCtl2	DT	DosDevIOCtl
DosPhysicalDisk	DT	DosPhysicalDisk

Code:

DT = API Parameter data type changes such as changing from USHORT to ULONG.

Table 1-22. *16-bit and 32-bit device management APIs.*

Table 1-23 shows the 16-bit and 32-bit device management APIs that have no equivalent mapping.

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosCLIAccess	
DosPortAccess	
DosR2StackRealloc	

16-BIT, NO 32-BIT API	32-BIT, NO 16-BIT API
DosCallBack	

Table 1-23. Device management APIs with no equivalent.

Pipe Management APIs

The pipe management OS/2 API prototypes and definitions are included by specifying INCL_DOSNMPIPES for named pipes and INCL_DOSQUEUES for unnamed pipes. These APIs are used for managing a pipe, which is a tool used for interprocess communication.

With the exception of a number of API name changes, these functions have remained relatively unchanged between the 16-bit and 32-bit version.

Table 1-24 shows the comparable 16-bit and 32-bit pipe APIs within the INCL_DOSNMPIPES and INCL_QUEUES groupings.

16-BIT API	CODES	32-BIT API
DosCallNmPipe	N,DT	DosCallNPipe
DosConnectNmPipe	N	DosConnectNPipe
DosDisConnectNmPipe	N	DosDisConnectNPipe
DosMakeNmPipe	N,DT	DosCreateNPipe
DosMakePipe	N,DT	DosCreatePipe
DosPeekNmPipe	N,DT	DosPeekNPipe
DosQNmPHandState	N,DT	DosQueryNPHState
DosQNmPipeInfo	N,DT	DosQueryNPipeInfo
DosQNmPipeSemState	N,DT	DosQueryNPipeSemState
DosSetNmPHandState	N,DT	DosSetNPHState
DosSetNmPipeSem	N,DT	DosSetNPipeSem
DosTransactNmPipe	N,DT	DosTransactNPipe
DosWaitNmPipe	N	DosWaitNPipe

Codes:

DT = API parameter data type changes such as changing from USHORT to ULONG.

```
N  = API name change.
```

Table 1-24. *16-bit and 32-bit pipe APIs.*

16-BIT TO 32-BIT CODE PORTING TIPS

With the introduction of the 80386 microprocessor came the ability to provide a 32-bit operating system. Although the 80386 microprocessor was available years before the first 80386 32-bit operating system, the 16-bit application programmers writing programs to run on the current release of OS/2 1.x realized the importance of keeping in mind the possibility of porting the 16-bit code to the 32-bit environment one day.

For instance, they understood a fundamental change would be the size of an integer. An integer is 2 bytes in a 16-bit architecture and 4 bytes in a 32-bit architecture. The OS/2 1.3 Toolkit rarely used the integer as a data type for a parameter or structure field and instead specified the size as a short or long integer which doesn't change across architectures.

The OS/2 Toolkit also provides data type definitions for all simple data types in the os2def.h include file. If these data types were used within the 16-bit program and had to change during the 32-bit application, then these changes would be provided by the 32-bit toolkit's os2def.h file. For instance, within os2def.h is the data type definition PSZ which is defined within the OS/2 1.3 Toolkit as:

```
typedef unsigned char far  *PSZ;
```

Iin the OS/2 2.1 Toolkit, this is defined as:

```
typedef unsigned char *PSZ;
```

Notice the **far** pointer-type keyword is no longer defined. In the 32-bit model, there is only one type of pointer, and the 16-bit pointer types **near** and **far** have no meaning. If you had declared in your program a variable **p** as **unsigned char far * p**, you would find that you have to change this declaration before the 32-bit compiler will successfully compile this declaration.

There are some fundamental tips that will help ease the porting process from 16-bit to 32-bit OS/2 C programs.

- ❑ As already mentioned, you can take advantage of the data types defined in os2def.h. If you use these in the same manner as the OS/2 APIs and structures, your program's functions and structures will port well with its OS/2 API usage. Especially use the data types and definitions that are specific to the 16-bit environment in os2def.h.

- ❑ If you were using integers in your code, consider changing them to SHORT or LONG integers to further specify exactly what the required size needs to be. This is meant for maximizing the program's working set and specifying exactly what the program requires.

- ❑ Compile your 16-bit code with the maximum warning message option set. Then, change the C code such that these warning messages do not occur. Once you are ready to compile your code with the 32-bit compiler, also set the maximum warning message option on. Removing these warnings increases the chances that subtle compiler differences will not affect how your application runs.

- ❑ If you have a large application to port, and you don't feel you have time to completely port the application all at once, consider first porting the code that would garner the greatest improvement in overall performance. Code that does intensive calculations would likely give the most improvement.

- ❑ Thunking from 32-bit code to 16-bit code is easier than thunking from 16-bit to 32-bit code. This is because the 32-bit compiler provides a mechanism for doing this where the 16-bit compiler does not. Keep this in mind, and structure your porting endeavor around this model.

- ❑ Take advantage of the DosQueryProcType 32-bit API to allow your application to call either a 16-bit or 32-bit function. Using this API, you could structure your code such that the only change would be to drop in the replacement 32-bit module. Your program would then be ready to run.

SUMMARY

This chapter introduced the OS/2 2.1 programming model. The model offers the development of 16-bit, 32-bit, and mixed 16-bit and 32-bit applications. The concept of thunking was discussed in connection with mixed 16-bit and 32-bit applications as the method for accomplishing this mixture.

Also discussed were the differences between the 16-bit and 32-bit APIs available within OS/2 2.1. The control program APIs have under gone a number of name changes. The semaphore, memory management, signal, and file functions were affected the most.

CHAPTER 2

Moving from DOS to OS/2 2.1

"A love for tradition has never weakened a nation, indeed it has strengthened nations in their hour of peril; but the new view must come, the world must roll forward."
— Sir Winston S. Churchill

INTRODUCTION

Software developers have always strived to make their jobs easier and faster. This has become increasingly difficult with the rising complexity of applications. The DOS programming environment, although seemingly boundless when it first emerged, places many limitations on the aspirations of designers and developers. In many cases, the DOS environment itself needlessly increases the difficulty of tackling the problems that lie at the heart of the intended application.

Many software developers, especially those in the entertainment field, have resorted to writing their own memory managers or supporting other memory standards to break free from the shackles imposed by the 16-bit DOS environment. This takes time and energy from the real program. There must be a better way.

The 32-bit OS/2 2.1 environment offers an evolutionary step to software development. Multitasking, memory management, interprocess communications objects (IPCs), and many other functions and capabilities are built into the operating system. Their presence frees application developers from having to spend time implementing the fundamentals of those features in their programs. They can instead exploit these built-in features and focus their energies tackling those problems their programs were meant to solve. This chapter discusses what you, as a DOS developer or someone new to the 32-bit environment, can expect and plan for when developing and porting applications to OS/2 2.1.

RUNNING DOS WITH OS/2 2.1

Even before you consider porting your application from DOS to OS/2 2.1 or before you build an OS/2 2.1 application from the bottom up, consider the advantages that simply having OS/2 2.1 as your operating platform will give to your existing DOS programs:

❑ The ability to run multiple DOS applications simultaneously. Your applications will not lie dormant when switched to the background. Even without being coded to take advantage of OS/2's multitasking APIs, each DOS session participates in the OS/2 2.1 multitasking environment.

❑ A base memory set of approximately 633K for each DOS session. The amount of available memory can be further increased to use expanded and extended memory. All this conventional memory is available, since your device drivers and the like are loaded above the 1MB mark in your program's address space. DOS can be loaded into high memory space too. You are not limited to the size of your physical memory. Each session, whether DOS or OS/2, can exceed the physical memory on its workstation by using the virtual memory capabilities in OS/2. Memory that exceeds your physical memory is stored in a swap file on a disk partition. This is true whether the code or data exceeds physical memory or because other programs are currently occupying physical memory.

❑ Protection between sessions. You can compile and test code, as well as use other programs, knowing that the chances of a memory violation or other heinous error bringing your workstation down are remote if not altogether eliminated by OS/2.

❑ Full-screen and windowed sessions. Whether your DOS program uses a character or graphical user interface, OS/2 2.1 allows you to run it in full-screen mode by itself or windowed on the OS/2 desktop with other windowed sessions and PM programs. As mentioned before, even while running in a full-screen session, your program still participates in the OS/2 multitasking environment. With windowed sessions, you can tile your windows to view more information and you can use the clipboard functions to copy data between your DOS, Windows, and OS/2 programs.

❑ Customizable settings. OS/2 2.1 provides many parameters to tune your DOS session according to your application's needs. Does it need more file handles? Complete control over your mouse pointer? Changes to your font size? Each DOS session has a group of settings that can be unique from other DOS sessions. You can even run individual AUTOEXEC.BAT programs for each DOS session in OS/2 2.1.

❑ Allow multiple DOS versions. Do you need to have different versions of DOS for different programs? You can have DOS 5.0 run in one session and PC-DOS 6.1 run in another. Your new version of DOS can be booted from either your A-drive or hard drive and can then be run in a session.

❑ Binary compatibility. You can run your DOS applications right out of the box. They do not have to be recompiled to participate in the OS/2 2.1 environment.

VIRTUAL DOS MACHINES (VDMs)

Although designed as a portable 32-bit operating system, the initial implementation of OS/2 2.x exploits the capabilities of the Intel 80386, 80486, and Pentium™ processors to provide you with protected DOS sessions in which to run the DOS environment and DOS programs.

These sessions are known as Virtual DOS Machines (VDMs), since they give the programs running in them the illusion that the program owns the entire machine—the memory, the keyboard, the video display—everything. OS/2 2.1 accomplishes this by using the virtual 8086 mode of the 80386 and higher processors. This gives it memory protection and the ability to terminate a session should the program running in it crash.

With OS/2 2.1, multiple DOS sessions can run concurrently. By default, they are executed in protected mode as a single-threaded application. However, beginning with OS/2 2.1, you can specify that your DOS application should run in a dual-threaded session. For a two-threaded DOS session, the first thread will service I/O requests and the second will allow the application to continue to receive timer tick interrupts and thus be able to update the video or audio with minimal interruption. This change helps multimedia applications run more smoothly in OS/2. For example, DOS applications with this setting enabled are not interrupted when they need to fetch data from a disk.

Executing a session in protected mode keeps its address space separate from other sessions; thus they are shielded from other DOS or OS/2 sessions. The chances of the program itself or others bringing OS/2 down are significantly reduced, if not obliterated altogether. Up to 256 DOS, Windows, and OS/2 sessions are supported, but for performance reasons, only 32 concurrently running sessions at most are recommended.

The single DOS compatibility box, in OS/2 1.x has been eliminated. With the DOS box, programs could not run in the background. Instead, they were placed in a suspended state when switched out of the foreground. The DOS box couldn't have hoped to support over 630K in memory or run in a windowed session. (In fact, it only supported 513K of conventional memory.) There was only one, which made it unique, but it couldn't match the VDMs, or DOS sessions, that OS/2 2.1 provides in either numbers or functionality.

Protected Mode

Protected mode offers many advantages over running your programs in real mode. With real mode on the 80386, the processor acts like a 16-bit machine, addresses correspond directly to real addresses (as opposed to being mapped), and there's no paging, virtual memory, or memory protection. If you run a DOS program on a 80386, this is what you're running in, but most DOS programs expect to run in real mode, so that's what the virtual 8086 mode provides. Virtual 8086 mode is a superset of protected mode. Your DOS applications essentially get a 32-bit environment, the mapping of physical addresses to virtual ones (increasing your address space), memory protection, and the ability for programs to own their own address space.

Both the extended memory (EMS) and expanded memory (XMS) standards are supported, specifically EMS version 4.0 and XMS version 2.0. OS/2 1.x users didn't have this ability. Utilizing EMS or XMS allows you to run your DOS programs that recognize and exploit these memory features. In addition, if your program uses the DOS Protected Mode Interface (DPMI), you can also access over 1MB of memory.

Virtual Device Drivers

VDMs use Virtual Device Drivers (VDDs) to map hardware I/O devices to the actual devices. If a program uses a standard DOS device driver, it can continue to use it, so long as this device is not shared across all the sessions in the operating system.

A communications port would be one such shared hardware interface. OS/2 loads VDDs above the 1MB mark in the process's address space, whereas DOS itself would load it below 1MB. With VDDs, multiple DOS sessions can access the printer and communications ports, hardware BIOS routines, and other devices.

The DOS environment provided by the VDM is compatible with DOS 5.0, but you can boot your own version as well off a floppy disk and have it run in the same operating environment as your other DOS, Windows, and OS/2 programs.

THOSE DOS SETTINGS

OS/2 2.1 will run your DOS programs, whether you developed them yourself or bought them from a store, without modification. However, for increased performance and/or usability, you will need to examine the DOS settings associated with the VDM running your program. Tuning them will give you increased satisfaction with your program.

Rather than expounding on the description and use of these settings (entire chapters of many OS/2 books have already dedicated themselves to this), the following is a list of some of the available settings:

- ❑ DOS_BACKGROUND_EXECUTION (ON or OFF)—If set to off, your program will be placed in a suspended state when switched from the foreground to the background.
- ❑ DOS_FILES (20–255)—Specifies the number of file handles your DOS session can use.
- ❑ DOS_VERSION—Lists the DOS versions compatible with your VDM.
- ❑ HW_NOSOUND (ON or OFF)—Turns sound on or off.
- ❑ HW_TIMER (ON or OFF)—Allows the program direct access to the hardware timing ports.
- ❑ IDLE_SECONDS (0–60)—Specifies the time before the operating system reduces the idle time of the program.

- ❑ IDLE_SENSITIVITY (1–100)—Specifies the time before OS/2 2.1 reduces the polling time of the processor for the session. Polling time is used to detect if the user is physically using the program in the session.

- ❑ KBD_ALTHOME_BYPASS (ON or OFF)—If set to off, the ALT-HOME combination will not switch from a windowed to full screen session or vice versa. Some DOS programs use this key combination for other functions.

- ❑ KBD_BUFFER_EXTEND (ON or OFF)—If set to on, the buffer used to capture keystrokes is increased, allowing you to type a little faster even when your system is busy.

- ❑ KBD_CTRL_BYPASS (NONE, ALT_ESC, or CTRL_ESC)—Similar to the KBD_ALTHOME_BYPASS setting, in that if ALT_ESC or CTRL_ESC is selected, those key combinations work as defined by the DOS program.

- ❑ KBD_RATE_LOCK (ON or OFF)—If set to off, the session cannot change the keyboard repeat rate.

- ❑ MOUSE_EXCLUSIVE_ACCESS (ON or OFF)—If set to on, your DOS program is given exclusive control of the mouse pointer. This will eliminate any mouse shadowing that may occur, where two pointers or a pointer and a cursor appear to move in tandem or in a bizarre waltz.

- ❑ PRINT_TIMEOUT (15–3600)—Specifies the time in seconds before OS/2 2.1 times out on a print job to the printer.

- ❑ VIDEO_FASTPASTE (ON or OFF)—If set to on, this specifies the speed at which input coming from devices or functions other than the keyboard (such as the clipboard) is increased.

- ❑ VIDEO_ONDEMAND_MEMORY (ON or OFF)—If set to on, the memory used for video-save is not allocated until it is needed by an application.

- ❑ VIDEO_RETRACE_EMULATION (ON or OFF)—If set to on, the session video is not retraced unless it is requested by the program.

- ❑ VIDEO_ROM_EMULATION (ON or OFF)—If set to on, programs can use video ROM functions, if they are supplied by your video ROM.
- ❑ VIDEO_SWITCH_NOTIFICATION (ON or OFF)—If set to on, OS/2 2.1 notifies your program when the user switches it from a windowed to a full screen session or from a full screen session to a windowed session.
- ❑ VIDEO_WINDOW_REFRESH (1–600)—Indicates the interval between video refreshes in tenths of a second.

EXTENDED AND EXPANDED MEMORY

There are three types of memory standards: conventional, extended (XMS), and expanded (EMS). Other types of standards exist as well, but these are the big guns. Basic DOS uses conventional memory, which ranges from 0 to 640KB. Users can place devices and other drivers into the upper memory area (UMA), which is located above 640KB but below 1MB.

To access memory past 1MB, Lotus, Intel, Microsoft, and AST created the XMS standard, which lets applications access memory beyond 1MB in protected mode. (Real mode is limited to 1MB.) The XMS standard gives guidelines to applications to allow them to migrate their code and data between conventional and extended memory. Extended memory starts at 1MB and extends up to the limits of the user's physical memory size.

The EMS Standard

Expanded memory also allows applications to go beyond 1MB. Its name originates from the memory expansion cards that users could add to 80286 machines. Code or data objects stored in the EMS are mapped to real mode addresses using page blocks in the UMA. Lotus, Intel, and Microsoft developed the EMS standard.

Because of this page mapping, EMS tends to be slower than XMS, which is why most DOS programs recommend the use of XMS. Windows 3.1 can support either XMS or EMS memory. By including the HIMEM.SYS driver in your DOMFIG.SYS file, you can specify XMS memory. Windows uses the EMM386.EXE software driver to support EMS by carving a specified range of XMS memory.

The DPMI Standard

Eleven companies developed the DOS Protected Mode Interface (DPMI) standard to allow DOS extenders to also address memory beyond 1MB and multitask their processes more reliably. OS/2 2.0 supported DPMI version 0.9 with a smattering of version 1.0 features. OS/2 2.1 was enhanced to support DPMI client specifications.

As mentioned earlier, OS/2 2.1 supports XMS version 2.0 and EMS version 4.0. DOS applications using XMS can access memory up to 16MB, and applications with EMS can go up to 32MB of memory. Those using DPMI can go up to 512MB, the maximum user addressable range in OS/2 2.1. You can continue to use these memory enhancers with your DOS programs. Your VDMs can mix and match these addressing techniques; one can use EMS, the other XMS.

However, you can discard these standards (including DPMI) altogether by moving to the 32-bit memory environment of OS/2. OS/2 2.1 provides virtual memory, which means that your program can access memory beyond the physical memory of your system. Best of all, your program doesn't have to know this. It's all built into the operating system.

WHY MOVE TO OS/2 2.1?

There are a number of reasons to move from DOS to OS/2 2.1. Perhaps the most beneficial is putting an end to memory segmentation. Programmers need not agonize over the creation and maintenance of memory segments any longer.

Keeping your application as a DOS program instead of moving it to an OS/2 application also means that it will continue to run as a single-threaded application in OS/2. (Some DOS programs can take advantage of a second thread to enhance their multimedia performance, but the user has to specify this setting.)

If you stay with a single-threaded design, you won't increase the complexity of debugging your application, but this also means that you won't be able to boost the performance or the responsiveness of your user interface. Moving your application to exploit the multi-threaded environment will increase your capabilities in the eyes of your users.

And why not exploit the increasing market share of 32-bit processors? Workstations with the Intel 80386 and 80486 processors are predominant. Memory is cheaper too. Users will have fast workstations with plenty of memory, and with that, they will want applications that exploit their investments.

CONVERTING FROM DOS TO OS/2

What factors should you consider when you decide to make the leap from DOS to OS/2 2.1?

- ❑ Moving from multiple programming memory models to one.
- ❑ Using memory objects instead of memory segments.
- ❑ Choosing from the variety of memory APIs.
- ❑ Breaching the 640K barrier with virtual memory.
- ❑ Exploiting your Intel 32-bit processors.
- ❑ Handling devices and exceptions.
- ❑ Juggling your functions with preemptive multitasking.
- ❑ Incorporating the use of IPC objects.
- ❑ Modularizing your code with dynamic link libraries.
- ❑ Interfacing with different file systems.

From the list, you can see that memory is a significant theme in your move from DOS to OS/2. It is also an area in which DOS programmers spend a large part of their development time.

By moving to the 32-bit environment of OS/2 2.1, you'll also make your code more portable to other platforms. Make the heart of your program generic to the 32-bit environment and add specific modules applicable to each platform.

For example, you could add two user interface modules to the main part of your program, one that exploits PM for OS/2 environments and another that is written for the Motif user interface for AIX and UNIX environments.

PROGRAMMING MEMORY MODELS

Programming in the 16-bit world of DOS, you could choose from a variety of memory models, depending on the size and constraints of your program:

- ❑ Small—Supports a 64K data segment and a 64K code segment. Near pointers are used for both the data and code.
- ❑ Medium—Supports a 64K data segment and multiple 64K code segments. Near pointers to data are still used.
- ❑ Compact—Supports multiple 64K data segments and one 64K code segment. This model is essentially the reverse of the medium model. Programs compiled with this model access data using far pointers, but the code is small enough to take advantage of the speed offered by near pointers.
- ❑ Large—Supports multiple 64K data segments and multiple 64K code segments. Far pointers are used for both data and code. While this model is necessary for big programs, the pointer manipulation required to use and access the code and data will reduce the performance of the program.
- ❑ Huge—Supports multiple 64K data segments and multiple 64K code segments, and it allows arrays to exceed 64K in size. Like the large memory model, programs can have larger amounts of code and data, but only with a corresponding decrease in speed and programming flexibility.

As a 32-bit operating system, OS/2 2.1 unifies all these memory models into one, the flat memory model; hence, all objects are near instead of far. No longer will you need to manipulate pointers solely to manage the size and location of your data structures or deal with 64K segments; thus, you can reduce the size and complexity of your programs. Whereas the malloc 16-bit C function call would allocate memory from the near heap, the malloc function in a 32-bit C compiler, such as the IBM C Set ++

version 2.0 compiler, allocates memory from a linear address space. No pointer manipulation between segments is required.

MEMORY SEGMENTS vs. MEMORY OBJECTS

Continuing the theme of the flat memory model, OS/2 2.1 has abolished memory segments, those infamous 64K chunks familiar to DOS and OS/2 1.x programmers. Memory objects now take their place, and these objects can range from 1 byte to 512MB. Instead of reserving 100K across two segments (one of 64K, the other of 36K) you can now use one memory object allocated for 100K. OS/2 will allocate the 100K in multiples of 4K pages. The uniform size of pages makes them easier for the operating system to manage.

Switching to the flat memory model will allow you to slash much of the segmentation management code from your program. By moving to the flat memory model in OS/2 2.1, you'll also wave a tearful good-bye to 16:16 pointer arithmetic and the loading of segment registers. (But you'll always have Paris.)

THE SMORGASBORD OF MEMORY APIs

To port your memory allocation and deallocation calls, change your 16-bit memory APIs to either call the C library functions of malloc and free or the OS/2 control program APIs of DosAllocMem and DosFreeMem with your 32-bit compiler. By using malloc and free, you retain portability between 32-bit operating platforms, since C compilers on operating systems like UNIX and AIX also use the same library function calls.

However, by using the DosAllocMem and DosFreeMem APIs, you will gain more control over how your program uses the 4K pages allocated to your memory object. (Pages are the basic building blocks of your memory objects.)

These APIs, including DosSetMem, also allow you to establish a guard page to help you reduce the amount of memory you need to allocate and commit to physical memory at one time. When a guard page is accessed, your program can recognize that more memory needs to be allocated.

The difference between allocating and committing memory is an important distinction. When you allocate memory, 4K pages are reserved in the linear address space of your process. However, the real storage, whether in physical memory or in a swap file, has not been assigned to the reserved space. Committing memory assigns the reserved space to real storage, allowing you to access and use the entire memory object or only that portion that was committed.

Postponing a Memory Commit

To save the memory in your program's working set, postpone committing one or more of the pages that are assigned to your memory object. You can leave all or just a portion of your allocated memory uncommitted until you call the DosSetMem API with the commit attribute. Afterwards, you can access and write to the memory object.

For the most part, use the DosAllocMem API to allocate and commit small, fixed-sized objects with one API call. This will reduce the number of memory APIs you need to call. Use the DosAllocMem and DosSetMem APIs to initially allocate but defer committing large, variable-sized memory objects. Committing later will reduce the memory load on your system. If you need more, you can always commit more pages out of your initial allocation. Note that the committed memory for an object cannot exceed the size of its allocated memory.

If you commit only part of your allocated memory, you will need to use a guard page. Typically, the guard page is the last page of your committed memory. Your application can specify this page with the DosSetMem API. When this page is referenced, a guard page exception is generated. If your application has an exception routine registered with the operating system, OS/2 will call it. Your routine can take over to react and recover from the exception. Recovery consists of committing more memory, if possible, and setting a new guard page. All this is transparent to the user.

Since memory objects must be allocated in 4K page blocks, a proliferation of small memory objects can waste a lot of space and quickly lead to fragmentation. If you are using many small objects, try suballocating your memory with the DosSubAllocMem API. Memory objects suballocated from larger memory pools are allocated in increments of eight bytes. As with regular memory allocation, memory pools may be initially uncommitted. Suballocation can then commit memory automatically as needed.

If you decide to use the DosAllocMem suite of API calls for your program, you can always change your memory management calls to use malloc in the future by insuring that you always commit your memory objects before using them. This means you have combined the DosAllocMem and DosSetMem calls. Figure 2-1 illustrates this process.

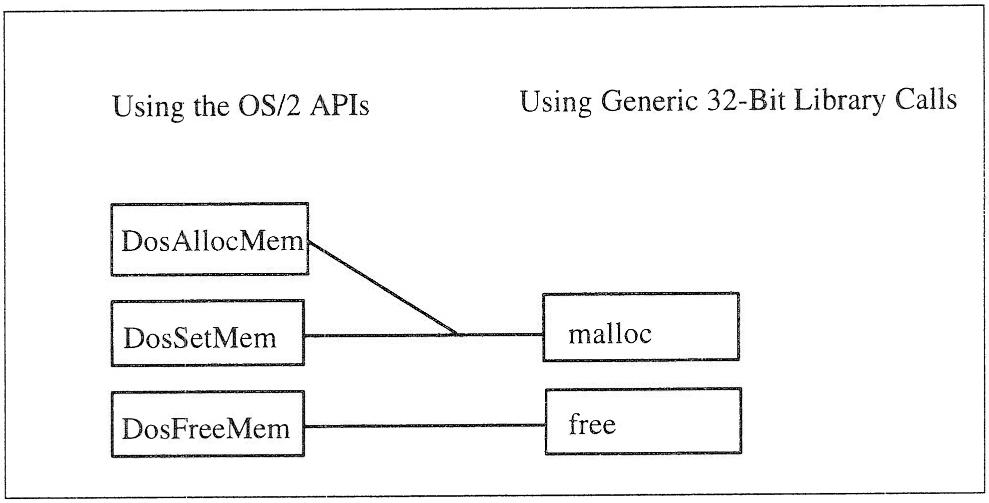


Figure 2-1. *Moving from OS/2 memory APIs to generic library calls.*

Be aware that the implementation of malloc and free may vary according to your compiler. Note also a caveat with the free library call. With the IBM C Set ++ version 2.0 compiler, the free function only returns memory back to the pool used by malloc and not to the operating system. Your application must call the IBM C Set ++ _heapmin API to return the memory back to OS/2.

VIRTUAL MEMORY

With 512MB available to a process, OS/2 2.1 has effectively removed any practical memory limitations your program may have. (OS/2 could have supported up to 4GB of linear address space and 64 terrabytes of non-linear address space, but for compatibility with OS/2 1.x 16-bit applications, OS/2 2.1 pegged its memory limit at 512MB.)

The simple rule here is to allocate memory when you need it and deallocate it when it is no longer needed or is no longer referenced. OS/2 2.1 will manage your workstation's memory by swapping 4K memory pages in and out of a disk swap file when demand on the physical memory grows and shrinks.

Of course, the vast majority of users will not have 512MB of real memory at their disposal. They will find a point in which their system thrashes. A thrashing system will constantly swap memory pages in and out of physical memory in an effort to service running tasks.

A thrashing workstation will quickly become unusable. Users have the responsibility to insure that they are not overloading their system by running too many programs or by not having enough physical memory. (Of course, developers have the responsibility to keep their applications lean and mean.)

You'll want to make judicious use of the memory that you've allocated (don't allocate 100K for a 10K array) and defer the commitment of large objects, but you will no longer have to worry about bumping into the 640K barrier.

32-BIT PROCESSOR

As a 32-bit operating system, OS/2 requires a 32-bit processor. For OS/2 2.1, this means an Intel 80386, 80486, or Pentium processor. (Intel 80386SX processors are supported too even though they have a 16-bit bus. 80386SX processors provide 32-bit addressing.) Using the OS/2 2.1 APIs, you indirectly get access to 32-bit processor instructions and arithmetic.

For the most part, these APIs will be transparent to you, except for a readily discernible increase in application performance. If you do not have a co-processor for your 80386 or 80486SX workstation, OS/2 2.1 uses its co-processor emulation functions. This allows you to incorporate Intel 80387 co-processor instructions in your program, if needed. (DOS applications running in a VDM get co-processor support only if they detect and utilize the co-processor.)

In addition, OS/2 2.1 has begun to exploit the new features of the Intel Pentium processor. The Pentium is backwardly compatible with the 80386 and 80486 processors, but when OS/2 2.1 recognizes that it is running on a Pentium, DOS sessions will take advantage of the processor's improved virtual 8086 support to boost the performance of DOS applications. With OS/2 2.1, you've embraced the Intel 80386, 80486, and Pentium platforms.

DEVICES

DOS provided only two types of devices: blocks (such as disks) and character (such as the keyboard, printer, and serial). Its base functions did not include graphics or mouse APIs. Programs had to directly access the hardware and the ROM BIOS. You still have these devices in OS/2 2.1, but they are considered virtual devices. These devices give each process the illusion that it possesses exclusive ownership of the devices. Virtual device drivers (VDDs) work with Physical Device Drivers (PDDs) to map an application's device call from its virtual to its physical implementation.

For example, a single virtual device driver for the COM port manages access to the port and allows all the DOS, Windows, and OS/2 sessions to access the COM and treat it as its own. Again, this functionality is built in, and you won't need to extend it unless you are writing a device driver yourself. (As mentioned earlier, your DOS application can still use its DOS device driver, so long as that driver is not shared across more than one session.)

EXCEPTION HANDLING

In DOS, if a memory error occurred, you probably lost control of your workstation and had to re-IPL. If a memory error occurred in Windows, you probably received a Universal Application Error or a General Protection Error. In OS/2 2.1, memory errors or other hard errors (known as *exceptions*) are now trapped at the process level. For VDMs, a VDM Manager traps an illegal operation so it won't affect the other VDM or OS/2 sessions.

In whatever session the error occurred, the operating system will generate a trap window, but the error will not bring your system to a sudden halt. Users can acknowledge the error and return to the operating system environment. Better yet, OS/2 developers can build functions into their applications to capture exceptions before they get to the user.

For example, if your application encounters an uncommitted memory exception, OS/2 2.1 will check to see if you have an exception handling function registered, and if so, it will call it, allowing you to commit the memory and continue.

MULTITASKING

DOS was designed as a single-user, single-task operating system. The Intel 8086 processor, on which DOS was designed, did not provide any program protection, nor did it expect or even envision programs to run concurrently in its domain.

In contrast, OS/2 2.1 has *preemptive multitasking* built into its operating environment. The processor still executes only one instruction at a time, but OS/2 divides the processor's time into slices, efficiently granting them to applications so they seem as if they execute concurrently.

The preemptive multitasking scheduler gives each task a CPU timeslice and will interrupt tasks at fixed intervals to insure that other tasks can run. The preemptive multitasking in OS/2 2.1 differs from cooperative multitasking in which an application must specifically relinquish its hold on the processor.

In a cooperative environment, an application can seize control of the CPU for as long as it wants, since there may be no guidelines as to when it must release its control, or the application may willingly choose to flaunt the guidelines that may exist.

A good example of tying up a single tasking or cooperative multitasking machine is to format a floppy disk. You may not be able regain control until the I/O operation completes.

In conjunction with the Intel 80386 and higher processors, OS/2 2.1 runs each process in a protected mode, which gives your application both the ability to participate in preemptive multitasking and the shielding of full memory protection.

By its nature, preemptive multitasking is more responsive, since it uses a timeslicing scheduler designed for the efficient use of your processor. Preemption removes your application from having to specifically code for relinquishing or re-wrestling control of the CPU.

Hierarchy of Multitasking Levels

OS/2 offers a hierarchy of multitasking levels for your applications. You can divide your application into sessions, processes, and threads and into priority levels. These levels can be modified dynamically. Sessions contain processes and each process has its own threads, file handles, and other objects. It also has its own address space. Each process has at least one thread, but additional threads can be added or deleted at your program's discretion.

With OS/2 2.1, a whole new way of designing has been opened up to you. If you already have a DOS application, think of where it spends its busy time. Is it when performing calculations, printing, or accessing the disk? Does the user have to wait until the application churns through CPU cycles to fulfill a task? If so, these are prime candidates for multithreading your application. When the user elects to use a processor-intensive action, think about creating a separate thread dedicated to the task. Doing so will allow you to return control back to the user via your primary thread.

A process can have up to 256 threads, but you are not limited to working with one process alone. There may be occasions when you have a task that needs to run in a separate process, but your main program may need to create it or maintain communications with it. Try spinning the task off as another process, either as a child in the same session or as a separate process in a different session altogether. Agent programs, such as ones that lie in the background collecting information, are well suited for the multitasking environment.

Ensuring Robustness

Understand that with the additional flexibility comes additional responsibility in the form of testing the robustness of your application. Threads share the memory of their parent process. If you have multiple threads running, are you serializing access to your process's memory objects? You'll need to prevent one thread from altering an object, when another thread may expect the old value in the memory object. Is the time it takes to create a new process or a new session necessary when a thread might execute the task just as easily?

Don't let these considerations deter you. Your program does not have to embrace the gamut of multitasking levels. The upshot is that you need not think of doing X and then Y to wind up at Z. You can now do X and Y together and get to Z a lot sooner for both your application and your users.

INTERPROCESS COMMUNICATION OBJECTS

If you decide to create other processes or if you want to interact with other processes that the user may be running, chances are that you'll need to use Interprocess Communication (IPC) objects. OS/2 2.1 offers a variety to choose from: shared memory, queues, pipes, and semaphores.

- ❑ Shared memory—While running, your application can dynamically allocate or request memory that can be shared with other processes. Programs that use shared memory must synchronize their access to it, and they can accomplish this with a semaphore. For example, process A first writes to shared memory M. Upon completion, process A would set semaphore X. Process B would detect that semaphore X was set, so it could read from shared memory M, knowing that the data in M was complete. Whereas private memory is allocated from the bottom of a process's address space, shared memory is allocated from the top, growing downward.

- ❑ Queues—Queues provide a means for ordering elements and storing them until the owning process can handle them. Any number of processes can write to the queue, but the process that creates the queue is the only one that can read or remove elements from the queue. The creating process can specify whether the ordering of elements on the queue should be first-in-first-out (FIFO), last-in-first-out (LIFO), or on a priority basis. LIFO queues act like stacks.

- ❑ Pipes—These IPC objects allow processes to communicate between each other with file handles and I/O calls. The data shunted through a pipe is stored in a shared memory area, unbeknownst to the user. Pipes can be named or anonymous. With named pipes, you could build an OS/2 application that converses with DOS applications or applications on machines across a LAN network.

- ❑ Semaphores—To insure serialized access for data or to synchronize functions between processes, use semaphores. They act like signal flags to stir waiting processes awake. Whether private or shared, they come in three flavors: mutual exclusion, event, and multiple event wait.

Note that some of these IPCs overlap in function. Internally, some are composites of two IPC objects. Use the ones that are right for your tasks.

DYNAMIC LINK LIBRARIES

When generating code with DOS compliers, you can compile your source code into either object (OBJ) files or executable (EXE) files. If you go the OBJ route to help modularize your code, you can gather together your OBJs into EXE or library (LIB) files. Library files can be linked together with your EXE when it is created. Figure 2-2 illustrates some of the possible routes to generating an EXE file with this process.

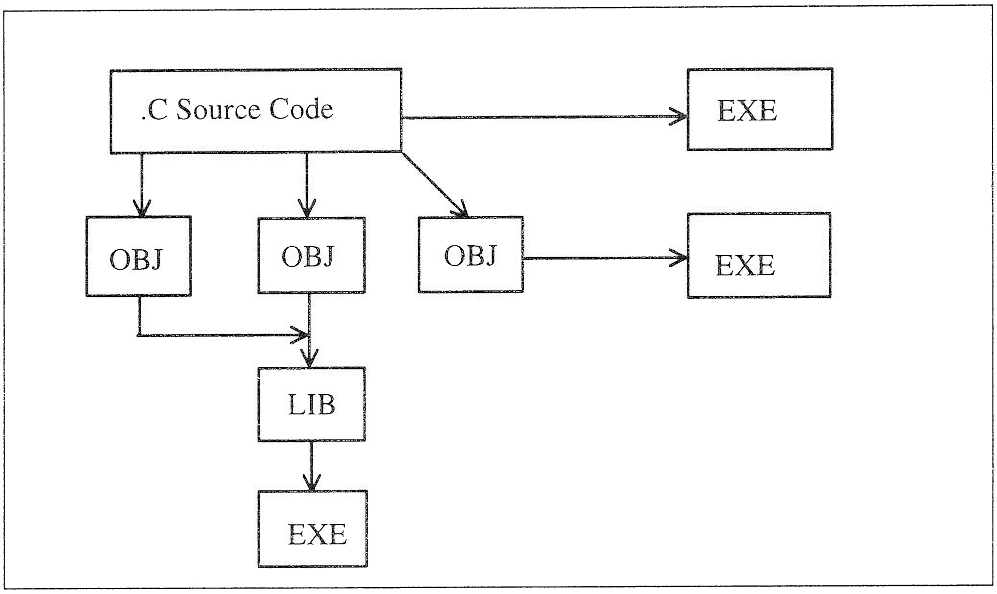


Figure 2-2. *Possible routes to generating an EXE file in DOS.*

The process of creating LIB files is known as static linking. However, linking with LIB files will increase the size of your generated EXE file, since the contents of the LIB file are essentially duplicated in the EXE file. OS/2 2.1 allows static linking and the creation of LIB files. It also allows dynamic linking. OS/2 2.1 compilers can combine OBJs to create Dynamic Link Library (DLL) files.

These files bind themselves to your EXE file, either when the EXE is invoked and loaded into memory or when the EXE specifically loads the DLL to call functions stored in it. The former route, known as *load-time dynamic linking,* resolves the external function references when your program is loaded into memory. APIs that are generic and that are used by a number of applications are usually stored in load-time

DLLs. The latter route, *run-time dynamic linking*, resolves your external references when your code is executed. Specialized DLLs and those whose loading can be deferred fall into the run-time category. You may want to defer loading a DLL if your program does not need it immediately.

Using DLLs will reduce the amount of memory required for your working set when your EXE is run. DLLs also provide their own initialization routines, are accessed without needing to switch segments (because there are none), and give you more flexibility in modularizing and packaging your code. You don't need to recompile your code to use them, and many programs can share the same DLL that has been loaded into memory.

For example, developers working on a database application could separate their client and server code into different DLLs, yet they could build only one EXE. The EXE file would call the appropriate DLL when needed. If the base code took 75K, client code 50K, and server code 125K, the base working set on your database client would only take 125K of memory (75K+50K) using DLLs. If you used LIB files only or combined all your OBJs into one gargantuan EXE file, it would take 250K (75K+50K+125K). Using DLLs will allow you to load only those modules that you need.

FILE SYSTEMS

Your programs will now need to contend with the possibility of the File Allocation Table (FAT) and High Performance File System (HPFS) file systems. OS/2 also allows Installable File Systems (IFS) for further expandability, but the more likely scenarios are FAT or HPFS. Users may install HPFS on one of their disk partitions for its improved file system performance. The OS/2 swap file or databases stored on an HPFS partition will benefit from increased performance.

DOS uses the FAT file system, so the naming syntax and file attributes of FAT files will be familiar to you. With the addition of HPFS, your application will need to recognize filenames that do not conform to the 8.3 specification. (The 8.3 specification refers to a filename up to eight characters long, followed by a period and an extension up to three characters.) Users can have longer names for both their file and directory names, and these names can include multiple periods and spaces.

Generally, if you do not plan to use extended attributes, the rest of your interface with the file system, whether FAT or HPFS, can remain the same. The file system APIs will shield you from the actual file system installed by the user. In fact, DOS programs can be stored on and run from HPFS partitions, even though they may run into trouble recognizing long filenames.

Extended Attributes

OS/2 2.1 also allows extended attributes to be stored with a file. These attributes allow you to store more information associated with the file, such as the font size that the user specified. On HPFS systems, these attributes are saved in a separate block with the file rather than in a separate INI type of file.

With C compilers for OS/2 2.1, you can use the standard C file I/O functions, such as fopen and fclose to access files. Your application can still use these, again for compatibility with other 32-bit platforms. However, to exploit the full gamut of OS/2 file functions, such as retrieving the date and time the file was last written to, you will need to use the OS/2 file APIs, such as DosOpen and DosClose.

Note that an enhanced version of FAT was introduced in OS/2 2.0 and carried over into OS/2 2.1. Not only is it compatible with the DOS FAT, but it also provides improved performance, greater caching, and the ability to *lazy write* data to disk.

STAYING WITH DOS

Let's say you don't want your application to exploit the 32-bit environment or the built-in capabilities that OS/2 2.1 offers. If so, then consider OS/2 2.1 or OS/2 for Windows as a development platform for your DOS applications. You can still run your DOS compiler in a VDM and it will still generate code compatible with the environment of DOS-only machines.

However, now you can do other things while your program is compiling and linking. Now you can work on other parts of your code in an editor in another session (DOS or OS/2), compute or conduct an analysis of values in a spreadsheet program (again, DOS or OS/2—even Windows), or check your mail and calendar functions, all while your program is compiling. You can run your executable program in a VDM for unit or functional testing purposes.

For your final suite of tests to insure that your program runs on a DOS-only machine, you can take your executable program to another workstation and run it. Better yet, set up your workstation with Boot Manager so you can boot either OS/2 2.1 or DOS when you start your machine or re-IPL.

In that way, you can use a single machine with either the OS/2 2.1 or exclusive DOS environment. You've lost nothing, protected your investment in DOS and Windows applications, and gained personal productivity just by developing your DOS applications on OS/2 2.1.

SUMMARY

OS/2 2.1 gives you an entirely new way to architect and develop your programs. Whether you choose to exploit the built-in features of OS/2 is up to you. Your programs can continue to run in DOS sessions or your DOS designs can be ported to OS/2 without much modification. In the latter case, you'll gain access to the speed of 32-bit APIs, but you'll also fail to take advantage of the true potential of your program.

Maybe you don't need to break your program into several threads. However, shared memory and other IPC objects may be not only attractive but also essential to the functions provided by your program. Perhaps you'll turn to OS/2 because its flat memory model makes your programming job simpler, your applications faster, and your product more portable to other 32-bit operating systems. OS/2 2.1 gives you the opportunity to make your applications as simple or as sophisticated as you like. It's your choice.

CHAPTER 3

Overview of the Control Program

"Good order is the foundation of all good things." — *Edmund Burke*

INTRODUCTION

This chapter provides a foundation for the next twelve chapters (4 through 15), which are concerned with present-day practices in application development. If you don't have to port old 16-bit applications to the 32-bit environment but aren't yet ready to leap into the LAN environment or into the new world of object-oriented programming, these are the chapters that will show you how to write today's OS/2 2.1 applications using Control Program APIs. The Control Program sits between users and their hardware, as shown in Figure 3-1. Sometimes called the supervisor or the core functions of OS/2 2.1, the Control Program consists of Layers 3, 4, and 5 in the diagram. The Control Program provides the APIs of the operating system.

ARCHITECTURAL LAYERS OF OS/2 2.1

The following discussion presents a simplified view of OS/2 2.1 architecture so that you can see where the Control Program fits into the whole system. There is a series of virtual layers in the Control Program that range, in general, from the layer closest to the user at the top of the diagram down to the layer farthest from the user at the bottom.

Layer 1: Users

Layer 1 in Figure 4-1 consists of various categories of end users, such as programmers, administrators, technicians, and office workers who use applications. End users, other than programmers, interact with the Control Program only by using the items contained in Layer 2.

1	Users
2	Workplace Shell Utilities Applications
3	OS/2 2.1 APIs (173 DosXxxxXxx Calls)
4	Dynamic Link Libraries (DLLs)
5a	OS/2 2.1 Kernel APIs & Working Routines
5b	Tasking, Dispatching, & Scheduling Routines
6a	Virtual Device Drivers
6b	Physical Device Drivers
7	Hardware

Figure 3-1*. Simplified architectural view of OS/2 2.1.*

Layer 2: Workplace Shell, Utilities, and Applications

Layer 2 consists of the Workplace Shell, the utilities, and the applications that users can start by clicking the mouse on an icon, for example, or by entering commands at the OS/2 2.1 command line. Users don't need to know about APIs to interact with the following elements at Layer 2:

- ❑ The Workplace Shell presents a Graphical User Interface (GUI) based on an object-oriented metaphor complete with a desktop, folders, files, and a trashcan.

- ❑ Utilities include such functions as backup and restore, FDISK, CHKDSK, and various system maintenance routines.

- ❑ Applications can include word processing, database, and communications programs, for example, as well as applications that you may write to provide a variety of services, such as sort, merge, and office mail.

To carry out requests that users make, the items at Layer 2 may need to use the API calls in Layer 3.

Layer 3: OS/2 2.1 APIs

Layer 3 contains the 173 OS/2 2.1 API calls whose names begin with the prefix *Dos* and continue with one or more descriptive, bi-capitalized words. They range alphabetically from DosAcknowledgeExceptionSignal to DosWriteQueue. Also known as Control Program routines or system calls, these APIs issue instructions to carry out the tasks that you code in an application. They exist either in the dynamic link libraries (DLLs) or in the kernel of OS/2 2.1.

APIs are provided for managing DLLs, for managing files and file systems, for multitasking, for interprocess communication, for error and exception handling, and for memory management, among other tasks. You can think of Layer 3 as a list of the 173 APIs that you can use in writing your applications. Such APIs as DosLoadModule, DosQueryModuleName, and DosFreeModule can be called to manage the code modules provided in Layer 4, although you don't need to load the Control Program DLLs explicitly.

Layer 4: Dynamic Link Libraries (DLLS)

Layer 4 provides dynamic linking to DLLs, which contain executable code modules that are available to an application at run time. An application loads a DLL only if it requires the APIs or code provided there. The application links to the DLL dynamically or at load time.

Other applications requiring those APIs can link to the DLL too. After the linking is accomplished and a DLL is in use, a need for multitasking or interprocess communication, for example, will require the use of the functionality of Layer 5.

Layer 5: The OS/2 2.1 Kernel APIs

Layer 5, the core or kernel of OS/2 2.1, contains all the multitasking components of the operating system and nearly all the other critical routines that keep the operating system going. You can think of these multitasking components as consisting of a higher level and a lower level, with the division indicated in Figure 3-1 by lines labeled 5a and 5b.

The higher level, 5a, contains the kernel APIs at the top, which control sessions, processes, and threads. The working routines at this level call the appropriate multitasking component to service a particular request. Components at this level include tasking, dispatching, and scheduling routines. The tasking routines manage the

APIs and the data structures related to processes and threads. The dispatching routines control when to switch contexts, that is, when to stop one thread and start another thread running. The scheduling routines manage the priority and the state of a thread and select the thread that will be started when the dispatching routines switch context.

The lower level, 5b, contains the exception, interrupt, and trap handling routines. Exception handlers manage unexpected events in the current process or current thread. Interrupt handlers manage the type of normal control switching caused by service requests from device drivers, such as an I/O request. Trap handlers provide a means of handling and routing exceptions, which usually stop all processing and notify the user or a program that an error has occurred. Intentional break points set for debugging purposes are also controlled by trap handlers, along with the OS/2 2.1 debugging API. The multitasking components at the lower level frequently need to interact with device drivers that act on behalf of the hardware.

Layer 6: Device Drivers

Layer 6 consists of device drivers at two levels, a virtual and a physical level, which control a piece of hardware. The virtual level, 6a, is provided for compatibility with the virtual machine concept in the DOS environment. This level also abstracts the devices so that you can plug in new devices with relative ease. The physical layer, 6b, is provided for basic device support, such as reading and writing to disk.

Layer 7: Hardware

Layer 7 includes the visible hardware that users are familiar with, such as the keyboard, system unit, monitor, and printer. Hardware also includes adapter cards, modems, plotters, tape drives, CDROMs, and other such devices.

FUNCTIONAL LAYERS OF THE CONTROL PROGRAM

Functional layers A through F in Figure 3-2 map to architectural layers 3 through 5 in Figure 3-1. These layers of Control Program functionality provide all the APIs necessary for writing applications to OS/2 2.1 except for device drivers, which are more complicated and require fuller treatment than this book can provide. In the discussion of each functional layer, a list is provided of the APIs available at that layer. Many of these APIs are explained further in Chapters 4 through 15, and code samples are included for many of the most widely used APIs. For complete reference information on these APIs, see the documentation supplied with the OS/2 2.1 Toolkit. Note that in the listings of APIs at each layer, the following information is provided:

❑ The name of the DLL in which the API is found.

❑ The name of the API.

❑ The ordinal number of the API. This is provided so that you can use the ordinal number instead of the name of API in your code. This feature is helpful when you want to link dynamically to an API in a DLL that you don't need to access frequently.

  Instead of loading the whole DLL statically, which can take some time, use the ordinal. Where 16-bit and 32-bit versions of an API exist, both ordinal numbers are provided. Where APIs are functionally equivalent except for a difference of name, the 16-bit APIs and their ordinals are included. Ordinals for functionally similar 16-bit APIs are also listed, but if no 16-bit equivalent exists, an x indicates this, as in ORD: x/195.

❑ A brief description of the API.

A	Dynamic Linking APIs
B	File, File System, and File Management APIs
C	Memory Management APIs
D	Multitasking (Program Execution Control) APIs: Sessions, Processes, and Threads
E	Interprocess Communication APIs: Semaphores, Pipes, Queues, and Timers
F	Exception Handling APIs: Error Handling, Exception Handling, Message Management, and Debugging

***Figure 3-2**. Functional view of the Control Program APIs.*

Layer A: Dynamic Linking

Most of the OS/2 2.1 operating system code modules are contained in dynamic link libraries (DLLs). Applications link to DLLs either dynamically at run time by means of the DosLoadModule call or statically by means of a linker. The linking combines modules and libraries, which have already been compiled, into executable code.

Dynamic linking APIs are found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of DLL APIs**
DosFreeModule ORD: 46/322	Frees the reference to the dynamic link module that was set for the current process.
DosFreeResource ORD: 208/353	Frees a resource previously loaded by the DosGetResource API.
DosGetResource ORD: 155/352 (DosGetResource2=207)	Returns the address of the specified resource object, which is a read-only data object accessible only at run time.
DosLoadModule ORD: 44/318	Loads a dynamic link module and provides a handle for it to the calling process.
DosQueryAppType ORD: x/323 (DosQAppType=163)	Returns the type, such as .EXE, .DLL, or device driver, of an executable file.
DosQueryModuleHandle ORD: x/319 (DosGetModHandle=47)	Returns the handle of a dynamic link module that was previously loaded to the calling process.
DosQueryModuleName ORD: x/320 (DosGetModName=48)	Returns the name of a handle for a dynamic link module; the name is fully qualified (drive, path, file name, and extension).
DosQueryProcAddr ORD: x/321 (DosGetProcAddr=45)	Returns the address within a dynamic link module for the procedure named in the call.
DosQueryProcType ORD: x/586	Returns the type (16-bit or 32-bit) of a callable procedure in a dynamic link module.
DosQueryResourceSize ORD: x/572	Gets the size of the specified resource object, which is a read-only data object accessible only at run time.

Layer B: Files, File Systems, and File Management

Files, file systems, directories, drives, disks, and file and device I/O have a large number of APIs because these are some of the most basic elements of an operating system. OS/2 2.1 provides both the File Allocation Table (FAT) and the High Performance File System (HPFS).

File APIs and those related to the directories, disks, and drives on which files reside are contained in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of File APIs**
DosCancelLockRequest ORD: x/429	Cancels a previously issued request to lock a range of a file with DosSetFileLocks.
DosClose ORD: 59/257	Closes a handle to a file; can also be used with device and pipe handles.
DosCopy ORD: 20/258	Copies a file or subdirectory to another location, known as the target file or subdirectory; does not erase the source file or subdirectory.
DosCreateDir ORD: x/270 (DosMkDir=66; DosMkDir2=185)	Creates a new directory or subdirectory; extended attributes can be set at the time of creation.
DosDelete ORD: 60/259	Deletes a file name from a directory but cannot delete read-only files; deleted files may be recovered by using the UNDELETE command if a storage directory was defined for the disk with the SET DELDIR command.
DosDeleteDir ORD: x/226 (DosRmDir=80)	Deletes a subdirectory from a disk; cannot delete the root or the current directory nor delete a directory unless it is empty.
DosDupHandle ORD: 61/260	Obtains a new handle for an open file and associates with it all the information in the original handle; cannot duplicate protected handles; also obtains handles for pipes.

API Name and 16/32-Bit Ordinal	**Description of File APIs**
DosEditName ORD: 191/261	Searches for and edits file names or directory names by transforming one name into another, such as changing from lowercase to uppercase; use of wildcard characters is permitted.
DosEnumAttribute ORD: 204/372	Searches for and identifies the name and length of extended attribute or a file or subdirectory.
DosFindClose ORD: 63/263	Ends a search by closing the handle to DosFindFirst or DosFindNext requests.
DosFindFirst ORD: 64/264 (DosFindFirst2=184)	Searches for and finds the first file or group of files that match the specified names; can also be used to find extended attributes.
DosFindNext ORD: 65/265	Searches for and finds the next file or group of files that match the name specified in a previous DosFindFirst; can also be used to find extended attributes.
DosForceDelete ORD: 203/110	Deletes a file name from a directory; deleted files are not recoverable.
DosFSAttach ORD: 181/269	Attaches or detaches a drive to or from a remote file system driver; can also attach or detach a pseudocharacter device name (that is, containing \DEV\) to or from a local or remote file system driver.
DosFSCtl ORD: 183/285	Provides an extended standard interface that an application can use to communicate with a file system driver.
DosMove ORD: 67/271	Moves a file to another location; can change the name of the file or can copy the file object to the same subdirectory and change its name.
DosOpen ORD: 70/273 (DosOpen2=95)	Opens a new or an existing file; the opened file can have extended attributes; returns a handle to the file; also opens pipes and devices.
DosQueryCurrentDir ORD: x/274 (DosQCurDir=71)	Obtains the full path name of the current directory except for the drive name.

API Name and 16/32-Bit Ordinal	**Description of File APIs**
DosQueryCurrentDisk ORD: x/275 (DosQCurDisk=72)	Obtains the current default drive of the process that issues this call.
DosQueryFHState ORD: x/276 (DosQFHandState=73)	Obtains the state of a file handle, such as its access mode and sharing mode.

DosQueryFileInfo
ORD: x/279
(DosQFileInfo=74;
DosQFileMode=75)
	Gets the following file information: Level 1, FILESTATUS3 data structure containing standard information. Level 2, FILESTATUS4 data structure containing the size of the extended attributes. Level 3, EAOP2 data structure containing additional extended attribute information.

DosQueryFSAttach ORD: x/277 (DosQFSAttach=182)	Gets information about an attached file system, either local or remote; can get information about a character device or pseudocharacter device attached to the file system.
DosQueryFSInfo ORD: x/278 (DosQFSInfo=76)	Gets information about a file system device, such as a disk or drive.
DosQueryHType ORD: x/224 (DosQHandType=77)	Finds whether a specified handle is for a file or a device.
DosQueryPathInfo ORD: x/22 (DosQPathInfo=98)	Gets information about a file or subdirectory similar to that found by DosQueryFileInfo, with the addition of Level 5, which contains extended attributes information.
DosQuerySysInfo ORD: x/348 (DosQSysInfo=66)	Gets the values of static system variables, such as the maximum length for a path name and number of DOS sessions allowed.
DosQueryVerify ORD: x/225 (DosQVerify=78)	Determines if write verification is active and enabled so that data written to disk is verified as being accurately recorded.
DosRead ORD: 137/281	Reads a specified number of bytes from a file to a buffer; can also be used to read from pipes and devices.

API Name and 16/32-Bit Ordinal	**Description of File APIs**
DosResetBuffer ORD: x/254 (DosBufReset=56)	Writes the buffers for a specified file to a storage device; can also be used with named pipes.
DosScanEnv ORD: 152/227	Searches for an environment variable (batch variable), such as PATH.
DosSearchPath ORD: 151/228	Searches for and finds files on a path that can be supplied by the process environment or by the caller.
DosSetCurrentDir ORD: x/255 (DosChDir=57)	Sets the current directory for the calling process.
DosSetDefaultDisk ORD: x/220 (DosSelectDisk=81)	Sets a drive as the default drive for the calling process.
DosSetFHState ORD: x/221 (DosSetFHandState=82)	Sets the state of the specified file handle for writes; can also be used for named pipes.
DosSetFileInfo ORD: 83/218 (DosSetFileMode=84)	Sets file information for Level 1 and Level 2 information as defined for the DosQueryFileInfo API.
DosSetFileLocks ORD: x/428 (DosFileLocks=62)	Locks or unlocks a range of an open file, depending on how the offset and length are set.
DosSetFilePtr ORD: x/256 (DosChgFilePtr=58)	Moves the read or write pointer from the beginning of the file, from the current location of the pointer, or from the end of the file.
DosSetFileSize ORD: x/272 (DosNewSize=68)	Changes the size of a file by truncating or extending its storage.
DosSetFSInfo ORD: 97/222	Sets information for a file system device, such as logical drive number.
DosSetMaxFH ORD: 85/209	Defines the maximum number of file handles available to the calling process when the system default of 20 needs to be increased.
DosSetPathInfo ORD: 104/219	Sets information for a file or directory, such as the level of information being defined (1 or 2).

API Name and 16/32-Bit Ordinal	**Description of File APIs**
DosSetRelMaxFH ORD: 108/382	Increases or decreases the maximum number of file handles for the calling process; saves handles currently open.
DosSetVerify ORD: 86/210	Sets write verification to active, which verifies that data written to disk is recorded correctly, or sets write verification to inactive.
DosWrite ORD: 138/282	Writes a specified number of bytes from a buffer to a file; can also be used to write messages to pipes.

Following are file system APIs that are new in OS/2 2.1. They are designed to provide a greater level of protection for a file handle than the similarly named APIs in earlier versions of OS/2. The main difference in the new APIs is an additional parameter, the file handle lock identifier, which contains a 32-bit address for the file handle.

On the DosProtect Open API, set the OpenMode parameter to specify a protected, instead of an unprotected, file handle. DosProtectOpen and all other DosProtectXxxxx APIs must then use the file handle lock ID parameter.

If a file is opened with the regular DosOpen API, the DosProtectXxxx APIs can still be used on the file by specifying a file handle lock ID of NULL.

Unauthorized processes can't access a protected file handle because they don't know the 32-bit address. Protected file handles can't be duplicated with DosDupHandle.

The new OS/2 2.1 protected handle APIS are found in the DOSCALLS DLL.

API Name	**Description of New OS/2 2.1 APIs**
DosProtectClose ORD: x/638	Closes a handle to a file; can also be used with pipes and devices.
DosProtectEnumAttribute ORD x/ 636	Identifies the names and lengths of the extended attributes for a file or subdirectory.
DosProtectOpen ORD: x/637	Opens a new or an existing file and returns a`protected file handle; the opened file can have extended attributes.

API Name	**Description of New OS/2 2.1 APIs**
DosProtectQueryFHState ORD: x/645	Queries and returns the state of a protected file handle.
DosProtectQueryFileInfo ORD: x/646	Gets the same file information as DosQueryFileInfo.
DosProtectRead ORD: x/641	Reads a specified number of bytes from a file to a buffer; can also be used with pipes and devices.
DosProtectSetFHState ORD: x/644	Sets the state of the specified file handle.
DosProtectSetFileInfo ORD: x/643	Sets Level 1 and Level 2 file information.
DosProtectSetFileLocks ORD: x/639	Locks and unlocks a range of an open file.
DosProtectSetFilePtr ORD: x/621	Moves the read or write pointer from the beginning of the file, from the current location of the pointer, or from the end of the file.
DosProtectSetFileSize ORD: x/640	Changes the size of a file.
DosProtectWrite ORD: x/642	Writes a specified number of bytes from a buffer to a file.

Layer C: Memory Management

Memory management involves allowing applications to allocate memory by units of 4KB pages in a 32-bit flat memory model. These units, or objects, can be in a *committed* state, in which physical storage has been allotted, or an uncommitted state, in which a range of addresses has been reserved but not yet assigned to physical storage.

Memory objects can be *private* or *shared*, and the latter can be protected from unauthorized use. Code and data needed for immediate processing are kept in memory and swapped out to external storage devices when memory is in an overcommitted state.

Since OS/2 2.1 memory management differs greatly from 16-bit memory management, there are few similarities between the 16-bit and 32-bit APIs. The nearest 16-bit equivalents are listed for your convenience. The memory management APIs are found in the DOSCALLS DLL.

API Name and 32-Bit Ordinal	**Description of Memory Management APIs**

DosAllocMem
ORD: x/299
(DosAllocSeg=34;
DosAllocHuge=40;
DosCreateCSAlias=43)
	Allocates a private memory object in the virtual address space of the process; reserves space for private pages, for example.

DosAllocSharedMem ORD: x/300 (DosAllocShrSeg=35)	Allocates a shared memory object in the virtual address space of the process; reserves space for shared pages, for example.
DosFreeMem ORD: x/304 (DosFreeSeg=39)	Frees a memory object, either private or shared, from the virtual address space of the process.
DosGetNamedSharedMem ORD: x/301 (DosGetShrSeg=36)	Gets access to a named shared memory object, which is an object whose name is preceded with \SHAREMEM\.
DosGetSharedMem ORD: x/302 (DosGiveSeg=37)	Gets access to a shared memory object, which is created with DosAllocSharedMem flags set for getting access.
DosGiveSharedMem ORD: x/303	Gives another process access to a shared memory object, which is created with DosAllocSharedMem flags set for giving access.
DosQueryMem ORD: x/306	Gets information about a range of pages in the virtual address space of the current rocess, including the base address, the size of the region, and the allocation flags.
DosSetMem ORD: x/305	Commits or decommits a range of pages in a memory object; can also change access protection for a range of pages, such as read and write access.
DosSubAllocMem ORD: x/345 (DosSubAlloc=147)	Allocates a block of memory (in multiples of 8 bytes) from a memory pool that was initialized by DosSubSetMem.
DosSubFreeMem ORD: x/346 (DosSubFree=148)	Frees a block of memory (in multiples of 8 bytes) that was allocated by DosSubAllocMem.

API Name and 32-Bit Ordinal	**Description of Memory Management APIs**
DosSubSetMem ORD: x/344 (DosSubSet=146)	Initializes a memory pool of at least 72 bytes for suballocation; can also be used to increase the size of a memory pool.
DosSubUnsetMem ORD: x/347	Ends the use of a memory pool and releases resources; must be used after DosSubSetMem.

Layer D: Multitasking

Multitasking provides a type of program execution control that will run multiple sessions, processes, and threads concurrently. A session, sometimes called a screen group, provides a space, such as a full screen, a windowed screen, or a DOS command line, where processes can be run. Devices, such as a keyboard and a mouse, are assigned to a session to create a logical workstation.

Each session can run a number of processes. A process, also known as a task or a program, is usually thought of as being part of an application and can own such resources as files, threads, and memory objects. Each process can run a number of threads.

A thread is the smallest unit of execution in a process and is usually equivalent to a piece of code that can run at the same time as the main program. Multitasking APIs are divided into those for sessions, processes, and threads.

Session APIs

Session APIS are found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of Session APIs**
DosSelectSession ORD: 9/38	Lets a parent session switch a child process to the foreground.
DosSetSession ORD: 14/39	Sets the status of a child session; status flags include selectable/not selectable and bond/no bond.
DosStartSession ORD: 17/37	Lets an application start another session and specify the name of the application to be started in the new session.

API Name and 16/32-Bit Ordinal	**Description of Session APIs**
DosStopSession ORD: 8/40	Ends one session or all sessions started by the calling process.

Process APIs

Process APIs are found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of Process APIs**
DosExecPgm ORD: 144/283	Lets a program request that another program run as a child process, synchronously or asynchronously; can let a thread run independently.
DosExitList ORD: 5/234	Keeps a list of routines to return control to when the current process ends.
DosKillProcess ORD: 10/235	Identifies a process to terminate and returns its termination code to the parent, if there is a parent process.
DosWaitChild ORD: x/280 (DosCWait=2)	Makes the current thread wait until an asynchronous child process ends and then returns process ID, termination code, and result code to the calling process.

Thread APIs

Thread APIs are found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of Thread APIs**
DosCreateThread ORD: 145/311	Creates a thread of execution under the current process; it runs asynchronously.
DosEnterCritSec ORD: 3/232	Disables the current process for thread switching; ensures that the current thread can continue processing.
DosExit ORD: 5/234	Lets a thread or process finish running and issues a completion code.
DosExitCritSec ORD: 6/233	Restores normal thread dispatching for the current process; used after DosEnterCritSec.

API Name and 16/32-Bit Ordinal	Description of Thread APIs
DosGetInfoBlocks ORD: x/312 (DosGetEnv=91; DosGetInfoSeg=8; DosGetPrty=9; DosGetPID=94; DosGetPPID=156)	Gets the address of the Thread Information Block (TID) of the current thread and the address of the Process Information Block (PID) of the current process.
DosKillThread ORD: x/111	Lets one thread end another thread in the current process; cannot end the current thread.
DosResumeThread ORD: 26/237	Restarts a thread previously halted with DosSuspendThread.
DosSetPriority ORD: x/236 (DosSetPrty=11)	Changes the priority of a child process in the current process; can also be used to change the priority of a thread.
DosSuspendThread ORD: 27/238	Delays execution of one thread in the current process until DosResumeThread is called to continue its processing.
DosWaitThread ORD: x/349	Makes the current thread wait until another thread in the current process ends and then returns the ending thread's ID.

Layer E: Interprocess Communication

Semaphores, pipes, queues, and timers are used for communications among running processes. Semaphores provide a method of synchronizing execution. Pipe, queue, and timer APIs can all use semaphores. Three types of semaphores are the following:

- ❑ Event semaphores—Threads and processes use these to signal other threads in the current process or another process that some event has happened, such as the release of a resource.

- ❑ Mutex (mutual exclusion) semaphores—Threads and processes use these to limit the interaction between two or more threads or processes. Mutex semaphores, for example, can be used when two processes are sharing a resource to ensure that only one accesses the resource at a given moment.

- ❑ Muxwait (multiple wait) semaphores—These are multiples of the other two types, such as several mutex semaphores combined into one. Muxwait semaphores let a thread or process wait for a set of event or mutex semaphores to finish. Muxwait semaphores also let a thread or process wait for any one of a set of event or mutex semaphores to finish. Note that you can't combine both event and mutex semaphores in a muxwait.

Semaphore APIs

Since the 16-bit and 32-bit semaphores are not functionally equivalent, only the 32-bit ordinals are provided here. Semaphore APIs are found in the DOSCALLS DLL.

API Name and 32-Bit Ordinal	**Description of Semaphore APIs**
DosAddMuxWaitSem ORD: x/341	Adds a mutex or an event semaphore to the list in a muxwait semaphore.
DosCloseEventSem ORD: x/326	Closes (ends access to) an event semaphore.
DosCloseMutexSem ORD: x/333	Closes (ends access to) a mutex semaphore.
DosCloseMuxWaitSem ORD: x/339	Closes (ends access to) a muxwait semaphore.
DosCreateEventSem ORD: x/324	Creates a named or unnamed event semaphore; can be private or shared, but a named event semaphore defaults to shared.
DosCreateMutexSem ORD: x/331	Creates a named or unnamed mutex semaphore; can be private or shared, but a named mutex semaphore defaults to shared.
DosCreateMuxWaitSem ORD: x/337	Creates a named or unnamed muxwait semaphore; can be private or shared, but a named muxwait semaphore defaults to shared.
DosDeleteMuxWaitSem ORD: x/342	Deletes a muxwait or an event semaphore from a list in a muxwait semaphore.
DosOpenEventSem ORD: x/325	Opens (gives access to) an event semaphore.
DosOpenMutexSem ORD: x/332	Opens (gives access to) a mutex semaphore.
DosOpenMuxWaitSem ORD: x/338	Opens (gives access to) a muxwait semaphore.

API Name and 32-Bit Ordinal	**Description of Semaphore APIs**
DosPostEventSem ORD: x/328	Posts an event semaphore; lets threads blocked by DosWaitEventSem resume execution.
DosQueryEventSem ORD: x/330	Gets the number of times that DosPostEventSem has been called since the event semaphore was last reset.
DosQueryMutexSem ORD: x/336	Gets the process ID and thread ID of the process that owns the mutex semaphore.
DosQueryMuxWaitSem ORD: x/343	Gets semaphore records from a muxwait semaphore list, such as whether the semaphores are shared or private and what the waiting policy is.
DosReleaseMutexSem ORD: x/335	Lets the thread that called DosRequestMutexSem release ownership of the semaphore.
DosRequestMutexSem ORD: x/334	Lets threads in the calling process request ownership of a mutex semaphore; threads in other processes must first call DosOpenMutexSem.
DosResetEventSem ORD: x/327	Resets an event semaphore, which blocks threads that subsequently call DosWaitEventSem.
DosWaitEventSem ORD: x/329	Lets a thread in the calling process wait for a semaphore to be posted; threads in other processes must first call DosOpenEventSem.
DosWaitMuxWaitSem ORD: x/340	Lets a thread in the calling process wait for a muxwait semaphore to be cleared; threads in other processes must first call DosOpenMuxWaitSem.

Timer APIs

Timers are services that provide access to the system date and time, set an interval for a thread to be suspended, and post an event semaphore at repeated intervals.

Timer APIs are found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of Timer APIs**
DosAsyncTimer ORD: x/350 (DosTimerAsync=29)	Starts an asynchronous timer that has a single interval; posts an event semaphore when the interval is over.
DosGetDateTime ORD: 33/230	Gets the current date and time as maintained by the operating system.
DosSetDateTime ORD: 28/292	Sets the current date and time that will maintained by the operating system.
DosSleep ORD: 32/229	Delays processing of the current thread for a specified interval.
DosStartTimer ORD: x/351 (DosTimerStart=30)	Starts an asynchronous timer that has repeated intervals; posts an event semaphore each time the interval occurs.
DosStopTimer ORD: x/290 (DosTimerStop=31)	Stops an asynchronous timer that has either a single interval or repeated intervals; used with both DosAsyncTimer and DosStartTimer.

Named Pipe APIs

A pipe is a method of communicating from one process to another by means of a system-maintained buffer that can be written to and read from as if it were a file. A named pipe can be used by any process that knows its name, whether related to the process that created the named pipe or not.

The named pipe APIs are found in the NAMPIPE DLL.

API Name and 16/32-Bit Ordinal	**Description of Named Pipe APIs**
DosCallNPipe ORD: x/240 (DosCallNmPipe=10)	Makes a procedure call to a pipe that handles duplex messages; opens, reads from or writes to, and closes the pipe.
DosConnectNPipe ORD: x/241 (DosConnectNmPipe=3)	Lets a server process place a named pipe into a listening state so that a client process can gain access to the pipe with DosOpen.
DosCreateNPipe ORD: x/243 (DosMakeNmPipe=1)	Creates a named pipe for the current process to use in communicating with a child process.

API Name and 16/32-Bit Ordinal	**Description of Named Pipe APIs**
DosDisConnectNPipe ORD: x/242 (DosDisConnectNmPipe=4)	Lets a server process acknowledge that a client process has closed the named pipe.
DosPeekNPipe ORD: x/244 (DosPeekNmPipe=7)	Lets a server process and its threads at the data in a pipe without disturbing or removing it.
DosQueryNPHState ORD: x/245 (DosQNmPHandState=5)	Gets information about the handle of a named pipe, such as whether the handle is for the server end or the client end and whether the pipe is a byte or a message type.
DosQueryNPipeInfo ORD: x/248 (DosQNmPipeInfo=2)	Gets information about a named pipe, such as the current and maximum number of instances and the pipe's name.
DosQueryNPipeSemState ORD: x/249 (DosQNmPipeSemState=14)	Gets information about local named pipes attached to a shared event or muxwait semaphore.
DosSetNPHState ORD: x/250 (DosSetNmPHandState= 6)	Resets the blocking mode and read mode of a mode of a named pipe; cannot change blocking to nonblocking if a thread is already blocked and cannot change the read mode of a byte pipe.
DosSetNPipeSem ORD: x/251 (DosSetNmPipeSem=13)	Attaches a shared event semaphore to a local named pipe, either a server handle returned by DosCreateNPipe or a client handle returned by DosOpen.
DosTransactNPipe ORD: x/252 (DosTransactNmPipe=9)	Writes to a duplex message pipe that is in message-read mode and then reads from it.
DosWaitNPipe ORD: x/253 (DosWaitNmPipe=8)	Enables a client process to wait for a named pipe instance to become available; the client process must also call DosOpen to get access to the pipe.

Unnamed Pipe API

In addition to named pipes, OS/2 2.1 provides one unnamed pipe API. An unnamed pipe can be used only by processes that are related to each other, such as a child process started by DosExecPgm from a parent process.

This API is found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	Description of Unnamed Pipe API
DosCreatePipe ORD: x/239 (DosMakePipe=16)	Creates an unnamed pipe for the current process to use in communicating with a child process.

Queue APIs

A queue is a means of communicating between two processes or threads by maintaining an ordered list of elements containing the information to be communicated. Queue APIs are found in the QUECALLS DLL.

API Name and 16/32-Bit Ordinal	Description of Queue APIs
DosCloseQueue ORD: 3/11	Lets the process that owns the queue delete the elements in a queue; lets the process that writes to the queue close access to the queue without deleting the elements.
DosCreateQueue ORD: 8/16	Lets a process create a queue and use it immediately.
DosOpenQueue ORD: 7/15	Lets a client process open (gain access to) a queue.
DosPeekQueue ORD: 5/13	Lets a server process and its threads look at a queue element without removing the element from the queue.
DosPurgeQueue ORD: 2/10	Enables a server process to purge all the elements from a queue without the possibility of recovery; client processes cannot use this API.
DosQueryQueue ORD: 4/12	Lets a server process and its threads get the number of elements in a queue; client processes can use this API after calling DosOpenQueue.
DosReadQueue ORD: 1/9	Lets a server process and its threads read and remove an element from a specified queue.
DosWriteQueue ORD: 6/14	Lets a server process and its threads write (add) an element to a queue; client processes can use this API after calling DosOpenQueue.

Layer F: Exception, Error, Debug, and Message APIs

Exceptions, errors, debugging, and messages all involve processing that is atypical in such a way as to cause some disruption of the expected. Exceptions are error conditions that usually terminate the processing of a thread. Errors and messages can range from simple informational postings to conditions that stop an application in its tracks. In debugging, you can set break points that stop processing at predetermined events. These APIs handle whatever error conditions that occur.

Exception APIs

Exception APIs are are provided only in the 32-bit versions of OS/2, where they are found in the DOSCALLS DLL.

API Name and 32-Bit Ordinal	**Description of Exception APIs**
DosAcknowledgeSignalException ORD: x/418	Lets thread 1 indicate that its process wants to receive additional signal exceptions.
DosEnterMustComplete ORD: x/380	Tells the system that a thread will enter a section of code in which signals and asynchronous process terminators will be delayed until the current thread completes.
DosExitMustComplete ORD: x/381	Provides an exit from a section of code called by DosEnterMustComplete.
DosRaiseException ORD: x/356	Lets a thread post a synchronous error that was held on a must-complete section of code.
DosSendSignalException ORD: x/379	Lets a process send an exception (Ctrl+C or Ctrl+Break) to another process to halt it.
DosSetExceptionHandler ORD: x/354	Lets a process register (add) an exception handler to manage errors that occur on the current thread.
DosSetSignalExceptionFocus ORD: x/378	Makes the current process the focus of Ctrl+C and Ctrl+Break signals in its screen group.

API Name and 32-Bit Ordinal	**Description of Exception APIs**
DosUnsetExceptionHandler ORD: x/355	Removes (unregisters) an exception handler from those associated with a thread.
DosUnwindException ORD: x/357	Calls and removes one or more exception handlers from a group of exception handlers associated with a thread.

Error APIs

Error APIs help manage error processing by categorizing errors, recommending actions to take, and deciding whether to inform end users that an error has occurred. Error APIs are found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of Error APIs**
DosErrClass ORD: 139/211	Classifies errors received from other APIs; tells where the error occurred and recommends recovery action.
DosError ORD: 120/212	Posts an error notice to the end user, or prevents a notice from being posted.

Debug API

The debug API lets you select and open a process for debugging and then examine and control the execution of that process. It is found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of Debug API**
DosDebug ORD: x/317 (DosPTrace=12)	Lets a calling process (a debug routine) control the execution of another process that controls debugging; uses semaphores to communicate between the two processes.

Message APIs

Message APIs help manage the storage, retrieval, and display of error messages to the end user.

They are found in the MSG DLL.

API Name and 16/32-Bit Ordinal	**Description of Message APIs**
DosGetMessage ORD: 2/6	Gets a message previously entered into a system message file in memory or storage; inserts variable text into the message.
DosInsertMessage ORD: x/4 (DOSInsMessage=3)	Inserts variable text into a message; does not retrieve the message but is used when messages are stored before variable text insertions are known.
DosPutMessage ORD:1/5	Sends a message to a file or device from which it can later be retrieved.

Miscellaneous APIs

In addition to the APIs associated with the kernel of OS/2 2.1, the operating system provides several miscellaneous APIs that pertain to devices or to national language and code-page considerations.

Device and Virtual Device Driver (VDD) APIs: These APIs interact with devices or virtual device drivers. Device and VDD APIs are found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of Device and Virtual Device Driver APIs**
DosBeep ORD: 50/286	Generates a sound at a specified pitch for a specified time period.
DosCloseVDD ORD: x/310	Closes (deletes access to) the handle of a specified virtual device driver; used by processes in protect mode.
DosDevConfig ORD: 52/231	Gets information about attached devices, such as ports, printers, drives, and monitors.
DosDevIOCtl ORD: 53/284 (DosDevIOCTL2=99)	Issues generic control functions to a device specified in the device handle, which must be open.
DosOpenVDD ORD: x/308	Opens a virtual device driver and returns a handle to it; used by processes in protect mode.

API Name and 16/32-Bit Ordinal	**Description of Device and Virtual Device Driver APIs**
DosPhysicalDisk ORD: 129/287	Obtains information about disks that can be partitioned, such as the number of partitioned disks on a system; returns a handle that can be used only by DosDevIOCtl.
DosRequestVDD ORD: x/309	Lets an OS/2 session in protected mode communicate with a virtual device driver; a request can be to read from or write to the VDD.
DosShutdown ORD: 206/415	Prepares for power to be turned off; prevents changes to file systems and writes system buffers to disk.

National Language APIs: These APIs let an application run independently of a national language, such as English, Spanish, or Chinese. A different code page, or translation table, is set up for each language, including double-byte character set (DBCS) languages like Japanese, Korean, and Chinese. The national language APIs help to manage the display of a particular language on the interface and in messages. National language APIs are found in the NLS DLL.

API Name and 16/32-Bit Ordinal	**Description of National Language APIs**
DosMapCase ORD: x/7 (DosCaseMap=1)	Maps the case of a string of binary values that represent ASCII characters; uses the case map in the country file, which defaults to COUNTRY.SYS.
DosQueryCollate ORD: x/8 (DosGetCollate=2)	Gets a collating sequence table from the country file; used in sorting the text of the national language.
DosQueryCtryInfo ORD: x/5 (DosGetCtryInfo=3)	Gets country-dependent information from the country file; used to format the national language.
DosQueryDBCSEnv ORD: x/6 (DosGetDBCSEv=4)	Gets a double byte character set environmental vector, such as the country code or the code page, from the country file.

Also, an API used for national language support is found in the MSG DLL:

API Name and 16/32-Bit Ordinal	**Description of National Language API**
DosQueryMessageCP ORD: x/8	Gets a message file that contains a list of code pages and national language identifiers.

In addition, two APIs used for national language support are found in the DOSCALLS DLL.

API Name and 16/32-Bit Ordinal	**Description of National Language APIs**
DosQueryCP ORD: x/291 (DosGetCP=130)	Lets a process get the name of its process code page, set by DosSetProcessCP, as well as the prepared system code pages.
DosSetProcessCP ORD: x/289 (DosSetProcCP=164)	Lets a process set its code page; used for printing text and does not affect keyboard or display code pages.

SUMMARY

This chapter introduces the Control Program, or the kernel, of OS/2 2.1 from both an architectural and a functional perspective. This chapter also introduces the APIs covered in the next twelve chapters and gives brief descriptions, the DLLs in which the APIs are located, and the ordinal numbers of the 173 APIs that comprise OS/2 2.1. Where matches occur, the 16-bit API ordinals are also included. Where functionality is similar, the nearest 16-bit equivalents are provided for your convenience.

CHAPTER 4

Compiling Programs and Creating Dynamic Link Libraries

"We shall not fail or falter; we shall not weaken or tire . . . . Give us the tools and we will finish the job." —*Sir Winston S. Churchill*

INTRODUCTION

If you are a DOS programmer, OS/2 will usher in a new programming model that will provide you with greater flexibility in how you design and package your code. When compiling in DOS, you can transform your C files directly into an executable file (EXE) or into object files (OBJs). These OBJ files can be linked together to form a library file (LIB) or an EXE. LIB files help store your function and procedure definitions until they are linked with other OBJ files and possibly other LIB files to create an EXE. With OS/2, you can continue to generate OBJ, LIB, and EXE files, but now you can also create Dynamic Link Library (DLL) files. DLLs will provide you with greater flexibility in modularizing your code. They will further help to reduce the size of your EXE by allowing you to divide your code into DLLs. When multiple copies of your EXE are running, DLLs will also help shrink the working set of memory required by the instances of your EXE during run time.

COMPILING

Unlike with an interpreted language, compilers take source code, written in a high-level language like C, and transform the code into machine code that is specific to an operating system or platform. The transformation from source to executable code can take one step, compiling directly to an EXE file, or two steps, compiling into OBJ files and then linking the OBJ files with other LIB and definition (DEF) files to create an EXE. The process of creating a DLL file is similar to creating an EXE file.

Note that the examples in this chapter, as well as those in the entire book, were developed with the IBM C Set ++ Version 2.0 compiler. Although the options may differ between OS/2 C compilers, the concepts will remain the same. If you own the IBM C Set/2 compiler, you should still be able to compile and run every example in this book without modification.

You can use the following files in your compilation process:

- ❑ .H—Header files store #defines, structures, and other commonly used definitions.
- ❑ .C—Source files contain the C language heart of your application.

The .C and .H extensions are the standard C naming conventions for these files. You can use other extensions, but for clarity and portability, you should use these and the other file extensions listed in this chapter.

The source and header files have a standard format as defined by the C syntax convention. In fact, with the IBM C Set ++ compiler, you can specify which syntax you want your source to follow, be it ANSI C, SAA, or migration.

If you are porting code from the IBM C/2 or Microsoft C 6.0 compiler to IBM C Set ++, use the migration language compilation option. This option will give you additional syntax flexibility during your conversion process.

Other files include the resource (.RC) and dialog (.DLG) files. These files are usually associated with Presentation Manager (PM) programs.

The SOM precompiler will also use source files with different extensions, such as .CSC.

Figure 4-1 on the next page illustrates one possible evolution from a source file to an executable file.

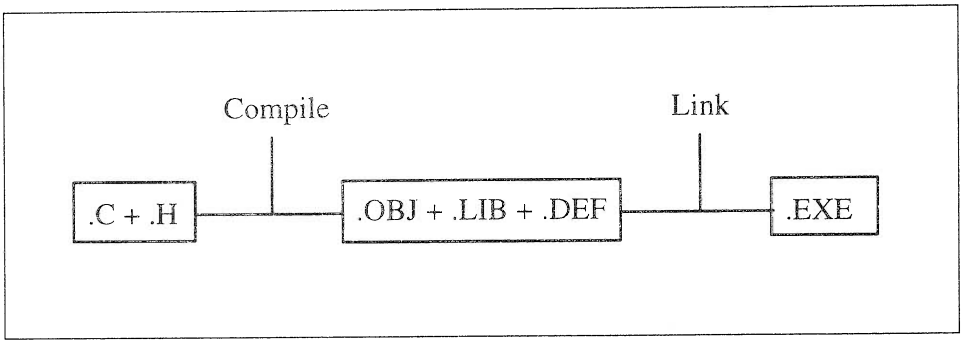


Figure 4-1. *The evolution of program files used in the compiling and linking process.*

Notice that the following files come into play during the linking process:

❑ .LIB—Library files help resolve the external definitions that your program has during the linking process.

❑ .DEF—Definition files store attributes about your program, such as its name and stack size.

The DEF files play an important part in creating your EXE and DLL files. You should have one for EXE files, but you must have one for your DLL file, since the DEF files will provide the roadmap that other programs will need when calling your DLL. Among other attributes, DEF files will expose the external functions in your DLL. If you don't have a DEF file for an EXE, your linker will use default values when creating your EXE.

When you compile your source, header, and DEF files, you can create a compiler listing file (.LST) of your source in addition to any OBJ or EXE files you may have generated. You can use this listing file along with your source code graphical debugger to assist you in debugging your source. Typically, you will not need to specifically generate this file as the proper compiling and linking options will enable your program for debugging.

There is a distinction between creating a regular EXE program and creating a subsystem program. EXE programs run in ring 3 of the operating system, which is where almost all user programs run. If you are writing a subsystem, such as a device driver, you would most likely run it at ring 0, at the heart of the kernel. This chapter focuses on creating EXE and DLL files to run at the user level. You will need to follow a different process to create a subsystem, since subsystems cannot use the default initialization, exception management, and termination routines that regular programs can. Almost all your applications will run at ring 3.

DEFINITION FILES (DEF)

DEF files for EXEs and DLLs can contain the following typical specifications. This list details the more widely used statements and attributes. Comments follow the semicolons in this list. Defaults are underlined.

- ❑ CODE [LOADONCALL | PRELOAD] ; This line states characteristics about your code. It determines whether the code segment gets loaded when accessed or when the program is initially invoked. Similar to an option used when loading data segments, LOADONCALL is the default.

- ❑ DATA [[MULTIPLE | SINGLE | NONE] [READWRITE | READONLY] [SHARED | NONSHARED] [LOADONCALL | PRELOAD]] ; If you want each process that uses the DLL to have a unique copy of the automatic data segments used by the DLL, specify MULTIPLE; otherwise, specify SINGLE. The default is SINGLE. (Note that data segments refers to the data area and not to those archaic 64K data segments mandated by 16-bit operating systems.)

  If you want programs to read and write to the DLL data segment, specify READWRITE; otherwise, specify READONLY. READWRITE is the default.

  If you want the READWRITE data area shared by all instances of the DLL, use the SHARED attribute; otherwise, use the NONSHARED attribute. SHARED is the default for DLLs. You may not want to share the data in your DLL, since the same DLL loaded in memory can be used by more than one program. Not sharing the data segments used by the DLL insures that the data is unique for each program that calls the DLL and that there is no chance the data will be overwritten by another program.

If you want to load the DLL's data segments when the functions that it contains are accessed, specify LOADONCALL; otherwise, specify PRELOAD to load the data when the DLL or EXE is loaded. LOADONCALL is the default.

- DESCRIPTION 'xxxx' ; Comment lines that you want to show up in the header portion of the EXE or DLL can go here. You may want to include your copyright statement here.

- EXPORTS ; A list of the externalized functions in the DLL. These functions are the ones that other programs can access. Note that you cannot export static functions. Only those functions that are specified with the extern keyword or that default to external functions can be exported. You can export functions and data. Specify your exported functions with the following syntax: name @ordinalNumber. If you specify a unique ordinal number with your name, calling programs can reference the function by either the name or the number.

- IMPORTS ; A list of the externalized functions that your EXE or DLL will use. If you use a LIB file, you do not need to use this statement to resolve your external references.

- HEAPSIZE xxxx ; Specifies the number of bytes reserved for the local heap.

- LIBRARY xxx [[INITINSTANCE | INITGLOBAL] [TERMINSTANCE | TERMGLOBAL]] ; Instead of using the NAME statement, DLLs use the LIBRARY statement. If you want to initialize the DLL each time it is loaded, specify INITINSTANCE, or else specify INITGLOBAL. TERMINSTANCE and TERMGLOBAL work similarly, except they execute termination rather than initialization routines. INITGLOBAL and TERMGLOBAL are the defaults if none are specified. Don't use the DLL extension as part of your DLL name for this statement.

- NAME xxxx [WINDOWCOMPAT | NOTWINDOWCOMPAT] [WINDOWAPI] ; EXE files use the NAME statement to identify their program. (DLLs would use the LIBRARY statement instead.) WINDOWCOMPAT indicates that the executable file can be run as an OS/2 full screen session or an OS/2 windowed session. Correspondingly, NOTWINDOWCOMPAT indicates that the executable file can run in an OS/2 full screen session only. WINDOWAPI indicates that the executable file is a PM program, since it will call PM APIs. You can use the EXE extension as part of your EXE name for this statement.

- ❑ PROTMODE ; This indicates that the program will run in protected mode. If not specified, the generated EXE or DLL file will be able to run in real or protected mode.

- ❑ STACKSIZE xxxx ; Specifies the number of bytes reserved for the program's stack.

CREATING AN EXE

The following program will choose the correct winning numbers for an upcoming lottery drawing. However, the date that these numbers are valid is a bit uncertain. (That is left for you to code.) It uses a mixture of C, DEF, and standard LIB files to create an EXE file. A make file kicks off the build process for the entire program.

The source file, shown in Figure 4-2, will seed the random number generator with the hundredths of seconds value before selecting the six winning numbers. These numbers will be unique and they will range between 1 and 50.

```
             /* c4_1.c - Pick the winning numbers.                  */

#define INCL_DOSDATETIME                    /* Date and time values     */
#include <os2.h>
#include <stdio.h>                          /* for the printf functions */
#include <stdlib.h>                         /* for the rand functions   */

#define TRUE  1
#define FALSE 0

VOID main (void)
{
   INT rnum;                                 /* random number           */
   ULONG seed;                               /* random number seed      */
   DATETIME  DateTime;                       /* Date and time structure */
   INT num[6];
   INT i,j,goOn;
   APIRET rc;

              /* Seed the random number generator                       */
   rc = DosGetDateTime(&DateTime);
   seed = DateTime.hundredths;
   srand(seed);
   printf("Random seed = %d.\n",seed);
                      /* initialize the numbers array                  */
   for (i=0;i<6;i++)
      num[i] = 0;

   /* Choose Numbers                                                    */
   for (i=0;i<6;i++) {
      do {
         goOn = TRUE;
         rnum = rand();
         if ((rnum > 50) || (rnum < 1))
             goOn = FALSE;
         else
            for (j=0;j<6;j++)
               if (num[j] == rnum)
                   goOn = FALSE;
      } while (goOn == FALSE);
      num[i] = rnum;
   }

                                  /* Print the results                  */
   printf("The winning numbers are: ");
   for (i=0;i<6;i++)
      printf("%d ",num[i]);
   printf("\nGood luck!\n");
}
```

Figure 4-2 *Lottery C language source file (C4_1.C).*

Although not needed for this program, the definition file in Figure 4-3 gives a name and a brief description to the EXE. It also indicates that the program will run in protected mode.

```
; Module definition file for C4_1.EXE

 NAME C4_1.EXE WINDOWCOMPAT

 DESCRIPTION 'Lottery Selection Program'

 PROTMODE
```

Figure 4-3. *Lottery definition file (C4_1.DEF).*

You can invoke the IBM C Set ++ compiler (ICC.EXE) and linker (LINK386.EXE) separately, or you can combine both commands into a make file, as shown in Figure 4-4. Using make files will make your development life easier. Not only are you saved from having to invoke the compiler and linker each time but, for large programs, the dependencies defined in a make file insure that only the changed or new files get compiled or linked. Start the make file for C4_1.EXE by entering the following command from your OS/2 prompt (the file is displayed after the command):

```
NMAKE C4_1.MAK
```

```
# C4_1.MAK

ALL:  c4_1.exe

c4_1.exe: c4_1.obj
   LINK386.EXE /DE /ALIGN:4 \
         c4_1.obj,c4_1.exe,,,c4_1.def

c4_1.obj: c4_1.c
   ICC.EXE /Sm /Ss /Q /Ti /W2 /C c4_1.c
```

Figure 4-4. *Lottery make file (C4_1.MAK).*

The winning numbers for one run are shown in Figure 4-5.

```
Random seed = 94.
The winning numbers are: 24 30 8 14 15 17
Good luck!
```

Figure 4-5. *Lottery output (C5_1.EXE output).*

Before Compiling

Before compiling and running your code, consider these environment variables and their purpose:

- ❑ PATH—Where your EXE and CMD files are searched for. REXX programs and OS/2 command files have the .CMD extension.
- ❑ DPATH—Where message files are searched for.
- ❑ INCLUDE—Where header files are searched for.
- ❑ LIBPATH—Where DLLs are searched for.
- ❑ LIB—Where IBM C Set ++ and Toolkit LIB files are searched for.

- ❑ TMP—Where temporary files generated and used by IBM C Set ++ are searched for. If not specified, IBM C Set ++ will usually default the temporary directory to the current directory. If you have a slower machine, set up a RAM or virtual disk to store temporary files. This will help shorten building time.

These variables are set in your CONFIG.SYS file. All, except LIBPATH, can be dynamically changed and customized for each OS/2 session. For example, you can add another directory to your PATH statement by entering the following from the prompt of an OS/2 session:

```
SET PATH=%PATH%;E:\NEWDIR\;
```

This change will be in effect as long as your OS/2 session remains active. To change your LIBPATH statement, modify its value in your CONFIG.SYS file and reboot your workstation.

A common technique is to place a period and a semicolon (.;) in front of your PATH and LIBPATH entries. This will insure that when you execute a program from your OS/2 command line prompt, it will search for the program and any associated DLLs from the current directory first before it proceeds down the remaining directory entries in the LIBPATH. Using .; will prevent you from loading a backlevel DLL from another directory if you keep different versions of the DLL in different directories.

The installation program for IBMCSet ++ and the OS/2 2.1 Toolkit will add their directories and their own environment variables to your CONFIG.SYS file. The OS/2 2.1 Toolkit contains the H, DEF, and DLL files that you will need to call the Control Program, PM, and many other APIs.

PASSING PARAMETERS

To pass command line variables and the environment variables to your program, declare your main procedure as follows:

```
int main(int argc, char **argv, char **envp)
```

The argc variable stores the number of parameters, including the program name itself. The argv variable stores a list of the parameters, and the envp variable stores a list of the environment variables. The IBM C Set ++ compiler creates initialization routines and includes them in your program to retrieve and store these values. If you have no need for command line parameters, forgo this overhead in both your program size and its initialization by declaring your main procedure as follows:

```
int main(void)
```

POPULAR COMPILER AND LINKER OPTIONS

Some of the more popular compiler options used by IBM C Set ++ are the following:

- ❑ /C+—Compile the source into an OBJ file. The default is to compile the C file and link the resulting OBJ file(s) to create an EXE. By deferring the linking until later, you have more flexibility on whether you want to mix and match the resulting OBJ files to create an EXE or DLL file.

- ❑ /G3—Generate code for the Intel 80386 processor. Use /G4 for Intel 80486 processors and /G5 for target workstations with the Intel Pentium processor. /G3 is the default. Using any of these flags will not prevent your code from running on any of the processors, but specifying the one that your program is likely to run on lets your program take advantage of any of the special features inherent in the respective processor.

- ❑ /Gd+—Dynamically links your code to the run-time libraries. For static linking, specify /Gd-. Static linking is the default For more on this, see the discussion later in this chapter on *Static and Dynamic Linking*.

- ❑ /Ge+—Compiles your code for an EXE file. This is the default option. If you want to create a DLL instead, use the /Ge- option.

- ❑ /Gm+—Use the multithreading libraries. The default is /Gm-, which indicates that the multithreaded libraries should not be used. Your program is a process that will always contain at least one thread, the main thread. Threads are the basic unit of execution in OS/2. If your process uses multiple threads, you will need to utilize the multithreaded libraries. These libraries include additional code not found in the single-threaded libraries to insure that the access to critical sections of the APIs in the libraries is serialized. With serialized access, only one thread can execute in the critical section at a time. Semaphores are used as stop lights to indicate if a thread is currently executing the code and whether a thread can begin executing the code. This prevents resources, such as a critical data structure, from being corrupted by another thread.

- ❑ /O+—Optimize your code. (The default is not to optimize, /O-.) As you are debugging your code, be sure not to include this flag with your compilation options. Otherwise, you may not be able to use your OBJs with the source code debugger. The same is true if you use another debugger. The execution of your optimized object file may not match the flow of code in your source file if this flag is used.

- ❑ /S[a|2|e|m]—Sets the language standard. /Sa indicates that you want your code to follow the ANSI standard. /S2 will force your code to conform to the SAA Level 2 standard. /Se allows your code to follow ANSI or SAA. /Sm provides for a more liberal code format for migration, such as from IBM C/2 or Microsoft C version 6.0. /Se is the default.

- ❑ /Ti+—Creates debugging information with your object files. As mentioned earlier, this is not recommended with the optimize option, /O+. The default, /Ti-, is not to generate this information.

- ❑ /Ss+—Lets you use double slashes (//) as a comment identifier before text on a single line in your source code. The default, /Ss-, does not allow double slashes as comment identifiers. The C++ language uses double slashes as comments. You may find them useful in your programs too, since using slash star and star slash (/* */) can be cumbersome at times.

- ❑ /W3+—Show the severe errors, regular errors, and warning messages generated as the compiler transforms your source file. There are different levels (0–3) you can specify to prune the (possibly) less important messages. You will want to use the highest option (/W3) at some point in time before you complete your program to insure that you have rooted out any potential errors, conflicts, inconsistencies, or both /W3 is the default.

For most options, the plus or minus sign following the option name is optional. If the option is specified without either sign, the compiler assumes that a plus follows the option. To reduce any confusion, when you browse your make files, you may want to include the + or - sign after the option.

Frequently used linking options are as follows:

- ❑ /ALIGNMENT—Specifies how the pages in the EXE should be aligned. You can specify any power of two from 2 to 32K. The default is 512. Exceeding 4K is not recommended, since OS/2 pages have a uniform length of 4K.

- ❑ /DEBUG—Prepares your EXE or DLL for use with the source code debugger. Your program can still run by itself, but it will contain extra information, such as symbolic data used by the debugger. To keep your file size down, be sure not to compile or link with the debugging options when shipping your final product.

- ❑ /EXEPACK—Compacts the pages in your EXE. This can make your EXE smaller.

- ❑ /STACK—Specifies the stack size of the EXE. The default is 8K, but you will want to increase it as the aggregate size of your automatic variables increases. (These variables are declared in forms like CHAR X[500] in your program.)

The IBM C Set ++ compiler program is called ICC.EXE and the OS/2 2.1 Toolkit linker is called LINK386.EXE. If you compiled your .RC into a .RES file, you would use the RC.EXE resource compiler. The source code debugger is called IPMD.EXE.

STATIC vs. DYNAMIC LINKING

To create EXE or DLL files, you will have to specify the LIB files that you wish to use. If you don't have any LIB files specified when you link your program, the LINK386 program will pick the default LIB files for you. If you need to specify different ones, the LIB files you choose will depend on whether or not:

- ❑ You have built custom-made LIB files using the IMPLIB program.

- ❑ Your target file uses multiple threads.

- ❏ You are building a user (ring 3) or a subsystem (ring 0) application.
- ❏ You will use the standard or migration code libraries.

You will also have to specify whether you want to link to the libraries statically or dynamically.

Static linking takes all the library functions that your application calls and links them into your EXE or DLL. In this case, the IBM C Set ++ functions would be placed in your EXE or DLL file. This will result in a larger target file. If a user can run more than one copy of your program or use more than one instance of your DLL, then there will be duplicate copies of the code in memory. This route has its benefits, since you can create a single EXE or DLL file that can be distributed without dependencies on a C library DLL, for example.

Dynamic linking does not merge the library functions called by your code into your EXE or DLL. Instead, the functions are loaded into memory either when the program is started or later when the program specifically loads the DLL and references the functions. Specifying dynamic linking will usually create a smaller target EXE or DLL file. By using DLLs, your program will resolve its external references at run-time.

To statically link your code, specify the /Gd- compiler option. Specify /Gd+ for dynamic linking. Statically linking your code is sometimes faster for the user, since all the code is loaded when the program is invoked, rather than waiting for when the functions are called. However, with dynamic linking, you can shrink your file size down and allow other programs to share the DLLs containing the library functions.

DYNAMIC LINK LIBRARIES (DLLs)

To reduce the complexity of developing large applications, you need to modularize your code. You will also want to draw upon those routines that have been robustly tested, that are shareable, and that hide their internals from other applications. You can think of DLLs as black boxes that not only have clearly defined inputs and outputs, but are reliable and shareable among a variety of applications. Programs, whether yours or others', cannot view the code or implementation of a function within a DLL. Only the APIs into the DLL are externalized. They are the only entry points into the DLL.

Some of the benefits of using DLLs are the modularity, ease of maintenance, and upgradability that they provide. For example, if you ship an EXE that can call five DLLs and if you need to change a function that resides in one of those DLLs, you will only need to recompile and relink the target DLL. This new DLL can be distributed to everyone using your EXE without their having to receive new copies of all your code. (This is providing that your external interface APIs haven't changed. For example, if you modified a function call to require four parameters instead of its original three or if you added a new function, you would have to modify, compile, and relink all the code that called those functions.)

You could build a program without DLLs, but the resulting EXE file will contain all the code for program, which could result in quite a large file size. DLL references can be resolved either at load-time, when the EXE is invoked, or at run-time, after the EXE was invoked and when specified by the program. Since DLLs can be loaded into memory at a later time, the physical size of the EXE can be reduced. The working set required by the EXE can be lowered since the EXE may not need all the DLLs during a single execution.

In summary, DLLs:

- ☐ Help modularize your code.
- ☐ Increase your code reuse.
- ☐ Can be shared among many programs.
- ☐ Assist in reducing the size of your program's working set.

You can create DLLs in the same fashion as creating an EXE. You will need to specify the /Ge- compiling option to tell the compiler that you want to create a DLL instead of an EXE. (The default is /Ge+, which means that the linker will create an EXE.) If your DLL allows multiple threads, you will also need to specify the /Gm+ compilation option, as described earlier.

There are various forms of DLLs. The first kind are the ones supplied with the C Set ++ library. If you specify the dynamic linking option when you link your EXE, your program will use the C library DLLs with the corresponding functions. The external references to IBM C Set ++ functions will be resolved at run time. The OS/2 APIs are stored in separate DLLs too. These DLLs, such as the DOSCALL1.DLL, are shipped with OS/2 and are stored in the \OS2\DLL\ directory. There are also DLLs without functions at all, like those created to store such resources as dialog boxes and messages. Lastly, you can create your own DLLs that can contain just your functions or a mix of your functions and C library functions.

CREATING A DLL

The following DLL example focuses on implementing functions for determining the average, high, and low stock prices for a company. An EXE will read in data for the Amber Waves Publishing Company, call the DLL to calculate the values, and report the results.

To build a DLL, you will need to:

- ☐ Create the C source and H header files.
- ☐ Create the module definition file (DEF).
- ☐ Compile and link the files to create a DLL.
- ☐ Create an import LIB file or use the original DEF file that other EXEs can use to know what functions it can link to and to resolve their external references.

Your C and H files contain the heart of your application. Your DEF file contains the characteristics of your DLL. See the previous section on DEF files for more information on the available options.

The first DLL example uses the make file in Figure 4-6. Note that for compiling, it uses the /Ge- option to create a DLL, /Gm- for the single-threaded libraries, and /Gd- for static linking to the libraries with the C functions that it calls.

The source could have been compiled with /Gd+ for dynamic linking. This would have resulted in a size savings for the EXE file. However, if you specify static linking, the C functions and variables used by your DLL become part of it. This helps it to become more self-contained, even if the code is replicated elsewhere in the calling program.

```
#  C4_2*.* calls a DLL, using DEF files.

ALL: stock.dll stock.lib c4_2.exe

# Create DLL and LIB files.

stock.dll: c4_2d.obj
   LINK386.EXE /DE /ALIGN:4 \
         c4_2d.obj,stock.dll,,,c4_2d.def

stock.lib: c4_2d.def stock.dll
   implib stock.lib c4_2d.def

c4_2d.obj: c4_2d.c c4_2d.h
   ICC.EXE /Sm /Ss /Q /Ti /W3 /C /Ge- c4_2d.c

# Create EXE file.

c4_2.exe: c4_2.obj
   LINK386.EXE /DE /ALIGN:4 \
         c4_2.obj,c4_2.exe,,stock.lib,c4_2.def

c4_2.obj: c4_2.c c4_2.h
   ICC.EXE /Sm /Ss /Q /Ti /W3 /C c4_2.c
```

Figure 4-6. *Make file (C4_2.MAK) for DLL and LIB files.*

Note also that if /Gm+ was specified for the multithreaded libraries, the resulting DLL would still function properly. However, since the program is only single-threaded, it will shoulder the performance overhead of the initialization and maintenance of code for multiple threads.

When linking, specify the created OBJ files, LIB files for your DLLs, and DEF file name. If you want to add additional library files or override any default library files, specify them with the linker as well.

The default _DLL_InitTerm function will be used when the DLL is loaded since another function wasn't substituted in its place. This function handles the default initialization of the DLL environment when the DLL is loaded. It also handles the termination of a DLL, freeing any allocated resources. You can override the default routines by creating your own _DLL_InitTerm function. You don't need to add any code to use the default function.

The source file for this first DLL example is shown in Figure 4-7. The C4_2D.C file will be compiled to create the STOCK.DLL file. The source file for the calling EXE is shown in Figure 4-8. The header file used by the DLL is shown in Figure 4-9. The header file used by the EXE is shown in Figure 4-10.

```
/* c4_2d.c  - will create the stock.dll file                      */

#include <stdio.h>
#include "c4_2d.h"


/* Find the highest stock price                                    */

void highPrice(STOCKPARMS stockparms, STOCKRETURN *stockreturn)
{
int i;

   stockreturn->date = stockreturn->price = 0;
   for (i=0;i<stockparms.numentries;i++) {
      if (stockreturn->price < stockparms.price[i]) {
         stockreturn->price = stockparms.price[i];
         stockreturn->date = stockparms.date[i];
      }
   }

}

/* Find the lowest stock price.                                    */

void lowPrice(STOCKPARMS stockparms, STOCKRETURN *stockreturn)
{
int i;

   stockreturn->date = stockparms.date[0];
   stockreturn->price = stockparms.price[0];
   for (i=1;i<stockparms.numentries;i++) {
      if (stockreturn->price > stockparms.price[i]) {
         stockreturn->price = stockparms.price[i];
         stockreturn->date = stockparms.date[i];
      }
   }
}


/* Find the average stock price                                    */

void averagePrice(STOCKPARMS stockparms, double *average)
{
int i,sum;
double avg;

   sum = 0;
   for (i=0;i<stockparms.numentries;i++)
         sum += stockparms.price[i];
   avg = (double) sum / (double) stockparms.numentries;
   *average = avg;
   return;
}
```

Figure 4-7. Stock functions DLL source file (C4_2D.C).

```
/* c4_2.c - calls a DLL, using LIB files.                           */

#define INCL_DOSFILEMGR                    /* File system values     */
#include <os2.h>
#include <stdio.h>
#include "c4_2.h"

VOID main (void)
{
HFILE          FileHandle;                  /* File handle to create   */
ULONG          Action;                      /* Results of action       */
UCHAR          filename[20], line[50];
ULONG          bytesRead;      /* number of bytes read from the file */
ULONG          filePtr;
BYTE           buffer[200];
int            i,j,index;                                /* counters */
int            date, price;
STOCKPARMS     stockparms;                   /* Main data structure */
STOCKRETURN    stockreturn;  /*returned structure from high/lowPrice */
double         average;        /* returned value from averagePrice */
APIRET         rc;                           /*    Return code       */

   strcpy(filename,"STOCK.TXT");
   rc = DosOpen(filename,                   /* Name of file to create  */
      &FileHandle,                          /* Address of file handle  */
      &Action,                              /* Pointer to action       */
      0,
      FILE_NORMAL,
      FILE_OPEN,
      OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE,
      NULL);

   if (rc == 0) {
      printf("Opened file %s.\n", filename);
   } else {
      printf("DosOpen failed for % with rc = %ld.\n",filename,rc);
      return;
   }

   /* Unnecessary, since initially opening the file will set
      the file pointer to the beginning, but used here as an
      illustration.                                                  */

   rc = DosSetFilePtr(FileHandle, 0, FILE_BEGIN, &filePtr);

   if (rc == 0) {
      printf("Set the file pointer to the beginning of the file.\n");
   } else {
      printf("DosSetFilePtr failed with rc = %ld\n", rc);
      DosClose(FileHandle);
      return;
   }

/* Read, display, and parse the contents of the data file          */
   rc = DosRead(FileHandle, buffer, sizeof(buffer),&bytesRead);

   if (rc == 0) {
      printf("Read %d bytes from the file.\n",bytesRead);
      printf("Contents:\n%s\n", (CHAR *) buffer);
      index = -1;
      i = 0;
      while (i<(bytesRead-1)) {
         j = 0;
         strcpy(line,"");
         while (buffer[i] != '\n') {
            line[j++] = buffer[i++];
         }
         line[j] = buffer[i];
         index++; i++;
         sscanf(line,"%d %d\n",&date,&price);
         stockparms.date[index] = date;
         stockparms.price[index] = price;
      }
      stockparms.numentries = index;
   } else {
      printf("DosRead failed with rc = %ld.\n", rc);
      DosClose(FileHandle);
      return;
   }

   rc = DosClose(FileHandle);

   if (rc == 0) {
      printf("Closed file %s.\n",filename);
   } else {
      printf("DosClose failed with rc = %ld.\n", rc);
   }

   /* Calculate and display values                                 */

   printf("Printing out values:\n");
   for (i=0;i<stockparms.numentries;i++) {
      printf("Date: %d Price: %d\n",stockparms.date[i],
         stockparms.price[i]);
   }

   highPrice(stockparms,&stockreturn);
   printf("High price on %d with value %d.\n",stockreturn.date,
      stockreturn.price);

   lowPrice(stockparms,&stockreturn);
   printf("Low price on %d with value %d.\n",stockreturn.date,
      stockreturn.price);
   averagePrice(stockparms,&average);
   printf("Average price value %f.\n",average);

   return;
}
```

Figure 4-8. *EXE calling the stock functions DLL.*

```
typedef struct _stockParms
{
   int  date[255];
   int  price[255];
   int  numentries;
} STOCKPARMS;

typedef struct _stockReturn
{
   int  date;
   int  price;
} STOCKRETURN;
```

Figure 4-9. *Stock functions DLL header file (C4_2D.H).*

```
typedef struct _stockParms
{
   int  date[255];
   int  price[255];
   int  numentries;
} STOCKPARMS;

typedef struct _stockReturn
{
   int  date;
   int  price;
} STOCKRETURN;

extern void highPrice(STOCKPARMS, STOCKRETURN *);
extern void lowPrice(STOCKPARMS, STOCKRETURN *);
extern void averagePrice(STOCKPARMS, double *);
```

Figure 4-10. *Calling stock functions with function names header file (C4_2.H).*

Now that you've created the DLL, you will need to create a LIB or DEF file that contains the "roadmap" to the externalized functions in the DLL. Creating a LIB file is the easier of the two routes.

To create a LIB file, use the IMPLIB utility program provided with the OS/2 2.1 Toolkit. It will extract information from the DEF file that you initially created for your

DLL. In turn, it will generate a LIB file that you can use with your linker that will resolve the external references in the program being linked. To create a LIB file for your previously created DLL, enter:

```
implib stock.lib c5_2d.def
```

Note that you need two DEF files. Figure 4-11 shows the DEF file for your EXE. Figure 4-12 shows the DEF file for your DLL.

```
; Module definition file for C4_2.EXE
NAME C4_2.EXE WINDOWCOMPAT

; You can specify the imported functions,
; but you don't need to since you have the C4_2D.LIB file
; for the DLL.

;IMPORTS
;   C4_2D.lowPrice ; Calculate the low stock price
;   C4_2D.highPrice ; Calculate the high stock price
;   C4_2D.averagePrice ; Calculate the average price
```

Figure 4-11. *Calling stock functions with function names definition file (C4_2.DEF).*

```
;Module definition file for STOCK.DLL

LIBRARY STOCK INITINSTANCE TERMINSTANCE
DATA MULTIPLE NONSHARED
CODE LOADONCALL
PROTMODE
EXPORTS
   lowPrice @1 ; Calculate the low stock price
   highPrice @2 ; Calculate the high stock price
   averagePrice @3 ; Calculate the average price
```

Figure 4-12. *Stock functions DLL definition file (C4_2D.DEF).*

To create a DEF file, you will need to list the external variables and functions in the DLL. This DEF file can then be used with your linker to resolve the external references in the DLL or EXE file being linked. The DEF file for this example EXE takes the following format:

❑ NAME C4_2 WINDOWCOMPAT ; C4_2.EXE is the name of your target program. WINDOWCOMPAT indicates that the file is compatible with the PM windowed environment.

- ❑ IMPORTS ; After this line, the external functions used by C4_2.EXE are listed in the format DLLname.function_name. Comments can follow the semicolon. If the program needed imported data structures, they would also be listed here. Since you are using a LIB file, you don't need to list your imported functions.

The DEF file for the DLL has the following format:

- ❑ LIBRARY STOCK INITINSTANCE TERMINSTANCE ; STOCK.DLL is the name of your DLL. Each instance of this DLL will have its own initialization and termination.
- ❑ DATA MULTIPLE NONSHARED ; The automatic data segments and the READWRITE data area will not be shared.
- ❑ CODE LOADONCALL ; The code segment will be loaded when accessed.
- ❑ PROTMODE ; The DLL runs in protected mode.
- ❑ EXPORTS ; After this line, the functions and data structures that the DLL makes known to the outside world are listed. Comments can follow the semicolon. STOCK.DLL exports only functions. If the program exported data structures, they would also be listed here.

After you invoke the C4_2.EXE program, it will read the contents of the data file, STOCK.TXT, and store it in a buffer. (STOCK.TXT is shown in Figure 4-13.) The data file has the format YYYYMMDD PRICE. YYYYMMDD corresponds to the year, month, and day of the closing stock price entry. YYYYMMDD also follows the International Standards Organization (ISO) date format.

Putting the date in this format makes it easier to sort. C4_2.EXE will close the file, parse the buffer, and put it in the STOCKPARMS data structure. Next, C4_2.EXE will call the functions in the STOCK.DLL to calculate the high, low, and average price of the stock.

```
19930101 100
19930102 90
19930103 95
19930104 100
```

```
19930105 110
19930106 100
```

Figure 4-13. *Stock prices data file (STOCK.TXT).*

```
Opened file STOCK.TXT.
Set the file pointer to the beginning of the file.
Read 83 bytes from the file.
Contents:
19930101 100
19930102 90
19930103 95
19930104 100
19930105 110
19930106 100
Closed file STOCK.TXT.
Printing out values:
Date: 19930101 Price: 100
Date: 19930102 Price: 90
Date: 19930103 Price: 95
Date: 19930104 Price: 100
Date: 19930105 Price: 110
High price on 19930105 with value 110.
Low price on 19930102 with value 90.
Average price value 99.000000.
```

Figure 4-14. *Calling stock functions with function names (C4_2.EXE output).*

USING THE C LIBRARY DLL CALLS

Behind many IBM C Set ++ library calls, there are one or more OS/2 APIs. This is true of the malloc and free C functions. They internally call the DosAllocMem and DosFreeMem APIs. Although using C functions may distance you from the OS/2 APIs, they offer both a higher level of abstraction and a chance to standardize your code for easier porting from another platform to OS/2 or from OS/2 to another platform. There are tradeoffs to using C functions instead of the APIs at their root. If you need a finer granularity of control for your tasks, use the APIs instead.

The C library functions of _loadmod and _freemod also help to abstract the OS/2 DosLoadModule and DosFreeModule APIs. The _loadmod function loads a DLL that you or another programmer created and returns a handle to the DLL. The _freemod function releases the handle that you previously loaded.

As with the DosLoadModule and DosFreeModule APIs, you would use the DosQueryProcAddr API to obtain the address of your desired function. Note that you do not have to use the LIB file for the DLL since you are dynamically loading it with the _loadmod function. You should not retrieve a handle to an IBM C Set ++ DLL using either the DosLoadModule API or _loadmod function. Specifying your linking options will insure that the appropriate DLLs are statically or dynamically linked for your program to run.

The next code example uses the same STOCK.DLL, but the main program, C4_3.EXE, has changed to explicitly load the DLL with the _loadmod C function, get the address of the functions with the DosQueryProcAddr OS/2 API function, and free the DLL handle with the _freemod C function. Since C4_3.EXE loads the DLL and retrieves the function addresses, it will not need a DEF file with the IMPORTS section.

_loadmod Function

```
_loadmod(
        char *module_name,
        ulong *module_handle);
```

Parameters:

- ❑ module_name (char) input—Filename of the DLL you wish to load. The program will search for the DLL in the directories specified by your LIBPATH environment variable.
- ❑ module_handle (ulong *) output—Handle to the loaded DLL, if the operation was successful.

_freemod Function

```
_freemod(
        ulong module_handle);
```

Parameter:

- ❑ module_handle (ulong) input—Handle to a previously loaded DLL. If successful, all program resources to the DLL are liberated back to the system.

DosQueryProcAddr API

After you load your target DLL with _loadmod, you will need to call the DosQueryProcAddr OS/2 API to retrieve the addresses of your functions and procedures in the DLL to use them.

```
          DosQueryProcAddr(
                           HMODULE ModuleHandle,
                           ULONG OrdinalNumber,
                           PSZ ProcedureName,
                           PFN ProcedureAddress);
```

Parameters:

- ❑ ModuleHandle (PHMODULE) input—Handle address to a previously loaded DLL.
- ❑ OrdinalNumber (ULONG) input—Ordinal number of the procedure address to be retrieved. If zero, the procedure name will be used for the search instead. To make your code more readable, use the procedure name. However, using ordinal numbers will find your function faster. If you are retrieving the address of a procedure in the DOSCALL*.DLL, you must use the ordinal number of the API. This number can be found in the OS/2 bseord.h file.
- ❑ ProcedureName (PSZ) input—Name of the procedure address to be retrieved. If an ordinal number is specified, this parameter is ignored.
- ❑ ProcedureAddress (PFN) output—Returned address of the procedure.

The source file for the C4_3.EXE program is shown in Figure 4-15. This program uses the C4_2D.H header file in Figure 4-9. The output is shown in Figure 4-16.

Note: The value of the DLL handle will differ from invocation to invocation.

```
/* c4_3.c - loads a DLL using the C library functions             */

#define INCL_DOSFILEMGR                  /* File system values     */
#define INCL_DOSMODULEMGR                /* DLL APIs               */
#define INCL_DOSMISC                     /* For DosSearchPath      */
#include <os2.h>
#include <stdio.h>
#include <errno.h>
#include "c4_2d.h"
```

```
VOID main (void)
{
HFILE       FileHandle;                 /* File handle to create   */
ULONG       Action;                     /* Results of action       */
UCHAR       filename[20], line[50], dllPath[256];
UCHAR       dllName[] ="STOCK.DLL";
ULONG       bytesRead;
ULONG       filePtr;
BYTE        buffer[200];
int         i,j,index;                                /* counters */
int         date, price;
STOCKPARMS  stockparms;                     /* Main data structure */
STOCKRETURN stockreturn;  /* returned structure from  high/lowPrice */
double      average;           /* returned value from averagePrice */
HMODULE     dllHandle;
PFN         faddr;                             /* function address */
APIRET      rc;                                /* Return code      */

   /* Get the handle of the DLL                                    */

   rc = DosSearchPath(SEARCH_ENVIRONMENT,"PATH",dllName,
          dllPath,sizeof(dllPath);

   if (rc != 0) {
      printf("Error:  Could not find %s.  DosSearchPath rc=%d",
         dllName,rc);
      return;
   }

   /*  Get the handle of the DLL                                   */

   rc = _loadmod(dllPath,&dllHandle);
   if (rc != 0) {
      printf("_loadmod failed for %s with rc = %ld.\n",dllPath,rc);
      if (errno == EMODNAME)
         printf("Module name is not valid.");
      if (errno == EOS2ERR)
          printf("System Error: %d."_doserrno);
      return;
   } else {
      printf("Loaded %s with handle %ld.\n",dllPath,dllHandle);
   }

   /* Open and read the data in the file.                          */
   strcpy(filename,"STOCK.TXT");
   rc = DosOpen(filename,               /* Name of file to create  */
      &FileHandle,                      /* Address of file handle  */
      &Action,                            /* Pointer to action     */
      0,
      FILE_NORMAL,                      /* Attribute of new file   */
      FILE_OPEN,
      OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE,
      NULL);
```

```
if (rc == 0) {
   printf("Opened file %s.\n", filename);
} else {
   printf("DosOpen failed with rc = %ld.\n", rc);
   return;
}

/* Read, display, and parse the contents of the data file          */

rc = DosRead(FileHandle, buffer, sizeof(buffer),&bytesRead);

if (rc == 0) {
   printf("Read %d bytes from the file.\n",bytesRead);
   index = -1;
   i = 0;
   while (i<(bytesRead-1)) {
      j = 0;
      strcpy(line,"");
      while (buffer[i] != '\n') {
         line[j++] = buffer[i++];
      }
      line[j] = buffer[i];
      index++; i++;
      sscanf(line,"%d %d\n",&date,&price);
      stockparms.date[index] = date;
      stockparms.price[index] = price;
    }
    stockparms.numentries = index;
} else {
   printf("DosRead failed with rc = %ld.\n", rc);
}

rc = DosClose(FileHandle);

if (rc == 0) {
   printf("Closed file %s.\n",filename);
} else {
   printf("DosClose failed with rc = %ld.\n", rc);
}

/* Calculate and display values                                    */

printf("Printing out values:\n");
for (i=0;i<stockparms.numentries;i++) {
   printf("Date: %d Price: %d\n",stockparms.date[i],
      stockparms.price[i]);
}

/* Retrieve function addresses.  The return code is checked for only
  the first DosQueryProcAddr call.                                 */

rc = DosQueryProcAddr(dllHandle, 0, "highPrice", &faddr);
if (rc == 0) {
   printf("Retrieved address for the highPrice function.\n");
   } else {
      printf("DosQueryProcAddr failed with rc = %ld.\n", rc);
      if (rc = _freemod(dllHandle)) {
         printf("_freemod failed with rc = %ld\n", rc);
      }
      return;
   }
   faddr(stockparms,&stockreturn);
   printf("High price on %d with value %d.\n",stockreturn.date,
      stockreturn.price);

   DosQueryProcAddr(dllHandle, 0, "lowPrice", &faddr);
   faddr(stockparms,&stockreturn);
   printf("Low price on %d with value %d.\n",stockreturn.date,
      stockreturn.price);

   DosQueryProcAddr(dllHandle, 0, "averagePrice", &faddr);
   faddr(stockparms,&average);
   printf("Average price value %f.\n", average);

   /* Free the DLL handle.                                            */

   if (rc = _freemod(dllHandle)) {
      printf("_freemod failed with rc = %ld\n", rc);
   }

   return;
}
```

Figure 4-15. *Calling stock functions with C functions source file (C4_3.C).*

```
   Loaded C4_2D.DLL with handle 3630.
   Opened file STOCK.TXT.
   Read 83 bytes from the file.
   Closed file STOCK.TXT.
   Printing out values:
   Date: 19930101 Price: 100
   Date: 19930102 Price: 90
   Date: 19930103 Price: 95
   Date: 19930104 Price: 100
   Date: 19930105 Price: 110
   Retrieved address for the highPrice function
   High price on 19930105 with value 110.
   Low price on 19930102 with value 90.
   Average price value 99.000000.
```

Figure 4-16. *Calling stock functions with C functions (C4_3.EXE output).*

USING THE OS/2 DLL API CALLS

The first DLL example, C4_2*, didn't load the DLL explicitly. Instead, the external functions were defined in the DEF file for the EXE. The second DLL example, C4_3*, did not need a DEF file for the EXE. It used a mixture of C functions and an OS/2 API function to load and free the DLL and retrieve the procedure addresses in the DLL.

This last DLL example will use the OS/2 APIs for DLLs exclusively. It will also use the ordinal number instead of the name to load the averagePrice function. Instead of _loadmod and _freemod, DosLoadModule and DosFreeModule are used. The C4_4* example also demonstrates:

- ❑ The DosQueryModuleName API to get the DLL name from a DLL handle.
- ❑ The DosQueryAppType API to retrieve information about the application (DLL or EXE) type.
- ❑ The DosQueryProcType API to retrieve information about the procedure type.

The DosQueryProcAddr API is carried over from the previous example.

The OS/2 DLL APIs are discussed in the following paragraphs. Note that the standard define and header include file entries for the OS/2 DLL APIs are as follows:

```
#define INCL_DOSMODULEMGR
#include <os2.h>
```

DosLoadModule API

To load a DLL and retrieve its handle, use the DosLoadModule API. The _loadmod function is the corresponding C function.

```
DosLoadModule(
              PSZ ObjectNameBuffer,
              ULONG ObjectNameBufferLength,
              PSZ ModuleName,
              PHMODULE ModuleHandle);
```

Parameters:

- ❑ ObjectNameBuffer (PSZ) input—Buffer in which error messages are placed should the DLL not be loaded. This buffer is used in a similar fashion to the ObjectNameBuffer in the DosExecPgm API.
- ❑ ObjectNameBufferLength (ULONG) input—Size of ObjectNameBuffer.
- ❑ ModuleName (PSZ) input—Name of the DLL to be loaded. Remember that the program will search for the DLL among the directories listed in the LIBPATH environment variable.
- ❑ ModuleHandle (PHMODULE) output—Handle address to the loaded DLL.

DosFreeModule API

To free your handle to a DLL, use the DosFreeModule API. The _freemod function is the corresponding C function. Note that this function frees the system resources used by the DLL if the usage count (number of processes using the DLL) is zero.

```
DosFreeModule(
               HMODULE ModuleHandle);
```

Parameter:

- ❑ ModuleHandle (HMODULE) input—Handle address to the DLL to be freed. This handle was previously retrieved with a DosLoadModule API call.

DosQueryAppType API

To retrieve the application type of an EXE or DLL, use the DosQueryAppType API:

```
DosQueryAppType(
                 PSZ ExecutableFileName,
                 PULONG ApplicationType);
```

Parameters:

- ❑ ExecutableFileName (PSZ) input—Name of the executable file that you want to query. Note that an "executable" file includes DLLs too. Although geared to querying the attributes of EXE and DLL files, you can specify a filename with any type of extension for this parameter.

- ApplicationType (PULONG) output—This parameter contains the attribute flags associated with the file. These flags compose the type of the application. They were retrieved from the application module header. Perform bitwise-and operations with the following values to determine if they exist.

Flag	**Value**	**Description**
FAPPTYP_NOTSPEC	(0x00000000)	Type not specified.
FAPPTYP_NOT_WINDOWCOMPAT	(0x00000001)	Not PM Window Compatible.
FAPPTYP_WINDOWCOMPAT	(0x00000002)	PM Window Compatible.
FAPPTYP_WINDOWAPI	(0x00000003)	Calls PM APIs.
FAPPTYP_BOUND	(0x00000008)	Bound by the BIND command.
FAPPTYP_DLL	(0x00000010)	A DLL.
FAPPTYP_DOS	(0x00000020)	In DOS format.
FAPPTYP_PHYSDRV	(0x00000040)	Physical device driver.
FAPPTYP_VIRTDRV	(0x00000080)	Virtual device driver.
FAPPTYP_PROTDLL	(0x00000100)	Protected memory DLL.
FAPPTYP_32BIT	(0x00004000)	32-bit executable file.

Note that this API is used by OS/2 and PM to determine the characteristics of the file being executed.

As mentioned earlier, if a DOS program is run from an OS/2 session, OS/2 can detect that the executable program is in DOS format. It will create a DOS session and run the program in it.

DosQueryModuleHandle API

To determine the handle of a previously loaded DLL, use the DosQueryModuleHandle API. This API is useful if you are unsure if the DLL has already been loaded. In a multithreaded application, a DLL could have been loaded by one of the threads at an earlier time. You can use this API to check if it was.

You cannot use this API to load the module, only to get its handle. Even though this API is not used in the example, it is worth describing as follows:

```
DosQueryModuleHandle(
                             PSZ ModuleName,
                             PHMODULE ModuleHandle);
```

Parameters:

- ❑ ModuleName (PSZ) input—Name of the DLL that was previously loaded or was suspected to be previously loaded.
- ❑ ModuleHandle (PHMODULE) output—Handle address to a previously loaded DLL.

DosQueryModuleName API

Since you may have specified only the filename of your DLL when you called the DosLoadModule API, you may want to determine the full pathname of the DLL. Do this with the DosQueryModuleName API. In your programming environment, you may have old versions of your DLL lying around in secluded directories. You can use this API to ensure that you are loading the most recent or the one you intended to load. The full pathname includes the drive, directory, and name of the DLL. The API is as follows:

```
DosQueryModuleName(
                         HMODULE ModuleHandle,
                         ULONG DLLBufferLength,
                         PCHAR DLLBufferName);
```

Parameters:

- ❑ ModuleHandle (PHMODULE) input—Handle address to a previously loaded DLL.
- ❑ DLLBufferLength (ULONG) input—Size of DLLBufferName.

- ❑ DLLBufferName (PCHAR) output—Buffer that will contain the fully-qualified DLL file name.

DosQueryProcType API

To determine if a procedure in a DLL is 16-bit or 32-bit, use theDosQueryProcType API. It is as follows:

```
        DosQueryProcType(
                         HMODULE ModuleHandle,
                         ULONG Ordinal,
                         PSZ ProcedureName,
                         PULONG ProcedureType);
```

Parameters:

- ❑ ModuleHandle (PHMODULE) input—Handle address to a previously loaded DLL.
- ❑ Ordinal (ULONG) input—Ordinal number of the procedure address to be queried. If zero, the procedure name will be used for the query instead.
- ❑ ProcedureName (PSZ) input—Name of the procedure to be queried.
- ❑ ProcedureType (PULONG) output—If 0 (PT_16BIT), procedure is of 16-bit type. If 1 (PT_32BIT), the procedure is of 32-bit type.

Example C4_4*

The source file for C4_4.EXE is shown in Figure 4-17. The output is similar to example C4_3*. It is shown in Figure 4-18.

```
/* c4_4.c - Calls a DLL using the OS/2 APIs.                              */

#define INCL_DOSFILEMGR                         /* File system values    */
#define INCL_DOSMODULEMGR                       /* DLL APIs              */
#define INCL_DOSMISC                            /* For DosSearchPath     */
#include <os2.h>
#include <stdio.h>
#include "c4_2d.h"  /* can use the H file from the previous example */
VOID main (void)
{
HFILE       FileHandle;                       /* File handle to create   */
```

```
ULONG       Action;                          /* Results of action     */
UCHAR       filename[20], line[50], dllPath[256];
UCHAR       dllName[]="STOCK.DLL";
ULONG       bytesRead;
ULONG       filePtr;
BYTE        buffer[300];
int         i,j,index;                                  /* counters */
int         date, price;
STOCKPARMS  stockparms;                      /* Main data structure */
STOCKRETURN stockreturn;  /* returned structure from  high/lowPrice */
double      average;           /* returned value from averagePrice */
HMODULE     dllHandle;
PFN         faddr;
CHAR        loadErrorStr[200];
CHAR        retDLLName[256];                   /* returned DLL name */
ULONG       dllType;                       /* Attributes of the DLL */
ULONG       procType;                 /* Attributes of the function */
APIRET      rc;                                /* Return code       */

   /*   Get the path to the DLL                                     */

   rc = DosSearchPath(SEARCH_ENVIRONMENT,"PATH",dllName,
         dllPath,sizeof(dllPath));

   if (rc != 0) {
      printf("Error:  Could not find %s.  DosSearchPath rc=%d",
         dllName,rc);
      return;
   }

   /* Get the handle of the DLL                                     */

   rc = DosLoadModule(
          loadErrorStr,
          sizeof(loadErrorStr),
          dllPath,
          &dllHandle);
   if (rc == 0) {
      printf("Loaded %s with handle %ld.\n",dllName,dllHandle);
   } else {
      printf("DosLoadModule failed for %s with rc = %ld.\n",
         dllPath,rc);
      printf("%s\n",loadErrorStr);
      return;
   }

   /* Although redundant, query the DLL name for illustrative
    purposes. Also, query application type information for the DLL.*/

   DosQueryModuleName(dllHandle,sizeof(retDLLName),retDLLName);
   printf("Status about %s.\n",retDLLName);
   if (rc = DosQueryAppType(dllName,&dllType)) {
       if ((dllType & FAPPTYP_PROTDLL) != 0)
          printf("   %s is a protected DLL.\n",dllName);
}

/* Open and read the data in the file.                                  */

strcpy(filename,"STOCK.TXT");
rc = DosOpen(filename,                    /* Name of file to create  */
   &FileHandle,                           /* Address of file handle  */
   &Action,                                 /* Pointer to action     */
   0,
   FILE_NORMAL,                           /* Attribute of new file   */
   FILE_OPEN,
   OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE,
   NULL);

if (rc == 0) {
   printf("Opened file %s.\n", filename);
} else {
   printf("DosOpen failed with rc = %ld.\n", rc);
   return;
}

rc = DosRead(FileHandle,buffer, sizeof(buffer), &bytesRead);

if (rc == 0) {
   printf("Read %d bytes from the file.\n",bytesRead);
   index = -1;
   i = 0;
   while (i<(bytesRead-1)) {
      j = 0;
      strcpy(line,"");
      while (buffer[i] != '\n') {
         line[j++] = buffer[i++];
      }
      line[j] = buffer[i];
      index++; i++;
      sscanf(line,"%d %d\n",&date,&price);
      stockparms.date[index] = date;
      stockparms.price[index] = price;
    }
    stockparms.numentries = index;
} else {
   printf("DosRead failed with rc = %ld.\n", rc);
}

rc = DosClose(FileHandle);
if (rc == 0) {
   printf("Closed file %s.\n",filename);
} else {
   printf("DosClose failed with rc = %ld.\n", rc);
}

/* Calculate and display values                                         */

printf("Printing out values:\n");
```

```
   for (i=0;i<stockparms.numentries;i++) {
      printf("Date: %d Price: %d\n",stockparms.date[i],
         stockparms.price[i]);
   }

   /* Checks could be done after other DosQueryProcAddr calls too   */
   rc = DosQueryProcAddr(dllHandle, 0, "highPrice", &faddr);
   if (rc == 0) {
      printf("Retrieved address for the highPrice function.\n");
      DosQueryProcType(dllHandle,0,"highPrice",&procType);
      if (procType == PT_32BIT)
         printf("   Function highPrice is 32-bit.\n");
      else
         printf("   Function highPrice is 16-bit.\n");
   } else {
      printf("DosQueryProcAddr failed with rc = %ld.\n", rc);
      rc = DosFreeModule(dllHandle);
      if (rc == 0) {
         printf("Freed %s handle %ld.\n",dllName,dllHandle);
      } else {
         printf("DosFreeModule failed with rc = %ld.\n", rc);
         return;
      }
      return;
   }

   faddr(stockparms,&stockreturn);
   printf("High price on %d with value %d.\n",stockreturn.date,
      stockreturn.price);

   DosQueryProcAddr(dllHandle, 0, "lowPrice", &faddr);
   faddr(stockparms,&stockreturn);
   printf("Low price on %d with value %d.\n",stockreturn.date,
      stockreturn.price);


   /* Call the ordinal number instead of the function name          */
   DosQueryProcAddr(dllHandle, 3, "", &faddr);
   faddr(stockparms,&average);
   printf("Average price value %f.\n", average);

   /* Free the DLL handle.                                          */
   rc = DosFreeModule(dllHandle);
   if (rc == 0) {
      printf("Freed %s handle %ld.\n",dllName,dllHandle);
   } else {
      printf("DosFreeModule failed with rc = %ld.\n", rc);
      return;
   }
   return;
}
```

Figure 4-17. *Calling stock functions with OS/2 APIs source file (C4_4.C).*

```
Loaded D:\C\CH4\STOCK.DLL with handle 3700.
```

```
Status about D:\C\CH4\STOCK.DLL.
Opened file STOCK.TXT.
Read 83 bytes from the file.
Closed file STOCK.TXT.
Printing out values:
Date: 19930101 Price: 100
Date: 19930102 Price: 90
Date: 19930103 Price: 95
Date: 19930104 Price: 100
Date: 19930105 Price: 110
Retrieved address for the highPrice function.
Function highPrice is 32-bit.
High price on 19930105 with value 110.
Low price on 19930102 with value 90.
Average price value 99.000000.
Freed D:\C\CH4\STOCK.DLL handle 3700.
```

Figure 4-18. *Calling stock functions with OS/2 APIs (C5_4.EXE output).*

TIPS

When compiling and linking your code and when creating DLLs, keep the following in mind:

- ❑ Use make files whenever possible to speed up your development process.
- ❑ Break your code into modules for maintainability and packaging. This includes not only separating your code into different C files, but also creating DLLs for related functions and to reduce overhead on the system.
- ❑ Dynamically link to C functions to conserve space. Statically link to reduce the number of files your program will require during run-time.
- ❑ Use the C functions to standardize your function calls for portability to other platforms. Use the OS/2 APIs for finer control.
- ❑ Use IMPLIB to create LIB files instead of using DEF files. You can use the generated LIB files when you link your programs. It's also easier for other developers to use a LIB file when linking code that will use your DLL.
- ❑ The /Ti+ compiler and /DE linking options allow you to create a debug version of your application. Make sure that you test your product on a non-debug or

"retail" version before releasing it to the world. Subtle timing differences between the debug and non-debug versions can reveal previously hidden errors. You may have to resort to the kernal debugger to find errors in your non-debug version. Don't just remove the options, recompile and relink, and send the product on its way.

- You should always check the return codes from your OS/2 API calls. To conserve space, not all the examples in this book do this. However, make sure that your program can handle an error return code.

API SUMMARY

The following lists the OS/2 Dynamic Link Library APIs. The DosFreeResource, DosGetResoure, and DosQueryResourceSize APIs were not covered in the chapter examples. They deal with resource objects, such as dialog boxes, that can be stored in an EXE or DLL.

- DosFreeModule—Frees a DLL and releases its handle.
- DosFreeResource—Frees the resource loaded by the DosGetResource API.
- DosGetResource—Gets the address of a resource in an EXE or DLL.
- DosLoadModule—Loads a DLL and retrieves its handle.
- DosQueryAppType—Retrieves the application type of a DLL or EXE.
- DosQueryModuleHandle—Retrieves the handle of a previously loaded DLL.
- DosQueryModuleName—Obtains the full path name of a previously loaded DLL.
- DosQueryProcAddr—Gets the address a function or procedure in a previously loaded DLL.
- DosQueryProcType—Determines if a procedure in a DLL is 16-bit or 32-bit.

- ❑ DosQueryResourceSize—Retrieves the size of a specified resource.

SUMMARY

If you are a DOS programmer, you will find that many of your compiling and linking techniques will transfer over to OS/2 2.1. If you are a Windows 3.1 programmer, the use of DLLs may be familiar to you. If so, you can carry your knowledge of DLLs over to OS/2. By exploiting the use of DLLs in your program, you will increase the modularity and reuse of your code.

CHAPTER 5

File Management and Use

"It is not enough to have a good mind. The main thing is to use it well."
— René Descartes

INTRODUCTION

This chapter shows you how to use various APIs to manipulate files and file systems. You can write applications to call a variety of file system APIs for file management and use.

This chapter discusses three types of file management APIs:

❑ APIs for managing and using files, such as creating, opening, reading, writing, and closing files.

❑ APIs for working with files as a whole, such as copying, moving, deleting, and searching for files.

❑ Miscellaneous APIs associated with the locations of files, such as paths, drives, and directories as well as APIs associated with file handles and file pointers.

APIs FOR MANAGING AND USING FILES

Five of the most common actions to take on files are performed by four file management APIs (DosOpen, DosRead, DosWrite, and DosClose). These APIs work with both the FAT and HPFS file systems. They are also used with pipes and queues.

Opening, Writing to, and Closing a File

The program in Figure 5-1 illustrates APIs for the following file management actions:

- ❑ Opening a file, or creating it if the file does not exist (DosOpen)
- ❑ Setting the file pointer to the end of the file (DosSetFilePtr)
- ❑ Writing a string to the file (DosWrite)
- ❑ Closing the file (DosClose)

Following are detailed descriptions of the APIs used in the program in Figure 5-1.

DosOpen

The DosOpen API creates a new file or opens an existing file, depending on the value passed in the the FileName parameter on the API call. The syntax of the DosOpen API follows:

```
DosOpen(
        PSZ     FileName,
        PHFILE  FileHandle,
        PULONG  ActionTaken,
        ULONG   FileSize,
        ULONG   FileAttribute,
        ULONG   OpenFlag,
        ULONG   OpenMode,
        PEAOP2  EABuf);
```

Parameters:

- ❑ FileName (PSZ) input—Indicates the address of the ASCIIZ path name of the file or device to be opened (or created if the file does not exist).
- ❑ FileHandle (PHFILE) output—Indicates the address of the file handle.
- ❑ ActionTaken (PULONG) output—Specifies the address of the variable that receives the value of the DosOpen action. This value is unpredictable if DosOpen fails. If DosOpen is successful, this value is one of the following:

Value	Name	Definition
1	FILE_EXISTED	The file could not be created because it already existed.
2	FILE_CREATED	The file was successfully created.
3	FILE_TRUNCATED	The file existed but was too long and was replaced with a file of a given length (truncated).

- ❑ FileSize (ULONG) input—When creating a new file or replacing an existing file, FileSize indicates the size of the file in bytes; it is ignored on other actions.

- ❑ FileAttribute (ULONG) input—Indicates the attribute bits of a file that can be set individually or in combinations:

Bit	Name	Value	Description
0	FILE_NORMAL	0x00000000	The file is open for reading and writing.
0	FILE-READONLY	0x00000001	The file is open for reading.
1	FILE_HIDDEN	0x00000002	The file is hidden; not displayed in a directory listing.
2	FILE_SYSTEM	0x00000004	The file is a system file.
3			Reserved; must be 0.
4	FILE_DIRECTORY	0x00000010	The file is a subdirectory.
5	FILE_ARCHIVED	0x00000020	The file is archived.
6-31			Reserved; must be 0.

- ❑ OpenFlag (ULONG) input—Specifies the action to take on an existing or non-existent file:

Bit	Value	Name	Description
0-3	0000	OPEN_ACTION_FAIL_IF_EXISTS	Open the file; fail if the file exists.

Bit	Value	Name	Description
	0001	OPEN_ACTION_OPEN_IF_EXISTS	Open the file if it exists.
	0010	OPEN_ACTION_REPLACE_IF_EXISTS	Replace the file if it exists.
4-7	0000	OPEN_ACTION_FAIL_IF_NEW	Open a file; fail if it doesn't exist.
	0001	OPEN_ACTION_CREATE_IF_NEW	Create the file if it doesn't exist.
8-31			Reserved; must be 0.

- OpenMode (ULONG) input—Specifies the mode, such as file access and file sharing:

Bit **Description**

0-2 Access mode flags show the actions your process can perform on the file:

Bit	Name	Value
000	OPEN_ACCESS_READONLY	0x00000000
001	OPEN_ACCESS_WRITEONLY	0x00000001
010	OPEN_ACCESS_READWRITE	0x00000002

3 Reserved; must be 0.

4-6 Sharing mode flags show the actions other processes can perform on the file:

Bit	Name	Value
001	OPEN_SHARE_DENYREADWRITE	0x00000010
001	OPEN_SHARE_DENYWRITE	0x00000020
010	OPEN_SHARE_DENYREAD	0x00000030
100	OPEN_SHARE_DENYNONE	0x00000040

7 Inheritance flag for file handles (not inherited by child processes):

	Bit	Name	Value
		OPEN_FLAGS_NOINHERIT	0x00000080
0	A process created by a call to DosExecPgm inherits the file handle.		
1	The current process has exclusive use of the file handle.		
8-10	These flags (locality of reference flags) show the way in which an application will gain access to the file:		

Bit	Name	Value
000	OPEN_FLAGS_NO_LOCALITY	0x00000000 Unknown locality.
001	OPEN_FLAGS_SEQUENTIAL	0x00000100 Sequential access.
010	OPEN_FLAGS_RANDOM	0x00000200 Random access.
011	OPEN_FLAGS_RANDOMSEQUENTIAL	0x00000300 Random access with some sequentiality.

DosSetFilePtr

The DosSetFilePtr API moves the pointer associated with reading and writing to the specified position in the file. The pointer is a signed 32-bit number. The syntax of the DosSetFilePtr API is as follows:

```
DosSetFilePtr(
              HFILE   FileHandle,
              LONG    lDistance,
              ULONG   ulMoveType,
              PULONG  NewPointer);
```

Parameters:

- FileHandle (HFILE) input—Contains the file handle returned by a previous call to DosOpen.

- Distance (LONG) input—Contains the offset in bytes showing where to move the pointer to. A negative value indicates to move backwards (toward the top of the file); a positive value indicates to move forward (toward the bottom).

- ❑ MoveType (ULONG) input—Specifies where in the file to start the move indicated in Distance; values include the following:

Value	**Name**	**Definition**
0	(FILE_BEGIN)	Start the move at the beginning of the file.
1	(FILE_CURRENT)	Start the move at the current location of the pointer.
2	(FILE_END)	Start the move at the end of the file. Setting the pointer at the end can also give you the size of the file.

- ❑ NewPointer (PULONG) output—Returns the address of the new pointer.

Note that if a file is shorter than the number of bytes specified in the Distance parameter, a program such as the one in Figure 5-1 will move the pointer to the end of the file and return the actual position in the NewPointer variable.

This technique is one way to determine the size of a file. Another is to move the file pointer from the end of the file (a negative offset, such as -500). If the file is shorter than 500 bytes, the file pointer is moved to the first byte in the file. The contents of the NewPointer variable will show the position of the file pointer in relation to the end of the file.

DosWrite

The DosWrite API writes the number of bytes you specify from a buffer to the file you name. The write starts at the current location of the file pointer. If the file is read-only, the write does not take place. The syntax of the DosWrite API follows:

```
DosWrite(
         HFILE    FileHandle,
         PVOID    BufferArea,
         ULONG    ulBufferLength,
         PULONG   BytesWritten);
```

Parameters:

- ❑ FileHandle (HFILE) input—Indicates the file handle obtained from a previous call to DosOpen.

- ❑ BufferArea (PVOID) input—Specifies the address of the buffer containing the data to write to the file specified in FileHandle.

- ❑ BufferLength (ULONG) input—Shows the number of bytes to write to the file.

- ❑ BytesWritten (PULONG) output—Indicates the address of the variable that will contain the number of bytes that were actually written to the file; useful to ensure that all the information you specified was written to the file.

The DosClose API closes a file handle. This API can also be used to close the handle to a pipe or a device. The syntax of the DosClose API follows:

DosClose

```
          DosClose(
                   HFILE FileHandle);
```

Parameter:

- ❑ FileHandle (HFILE) input—Indicates the file handle returned from a previous call to DosOpen (also used with handles returned from DosCreateNPipe, DosCreatePipe, and DosDupHandle).

The program in Figure 5-1 opens a file named filecode.txt or creates the file if it does not exist. The program sets the file pointer to the end of the file, writes a text string, and closes the file.

```
/* PROG5F1.C.  This program opens a file named filecode.txt        */

#define INCL_DOSFILEMGR                    /* File system values      */
#include <os2.h>
#include <stdio.h>

VOID main (USHORT argc, PCHAR argv[])
{
   HFILE   FileHandle;                     /* File handle to create   */
   ULONG   Action;                           /* Results of action     */
   APIRET  rc;                                    /* Return code      */
   UCHAR   buffer[100], filename[20];
   ULONG   bytesWritten;
   ULONG   filePtr;
   strcpy(filename,"FILECODE.TXT");
   rc = DosOpen(filename,                 /*  Name of file to create */
      &FileHandle,                        /*  Address of file handle */
      &Action,                            /*  Pointer to action      */
      0,                                  /*  Size of new file       */
      FILE_NORMAL,                        /*  Attribute of new file  */
      FILE_OPEN | OPEN_ACTION_CREATE_IF_NEW,
      OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE, NULL);

   if (rc == 0) {
      printf("Opened file %s.\n", filename);
   } else {
      printf("DosOpen failed with rc = %ld.\n", rc);
      return;
   }

   rc = DosSetFilePtr(FileHandle, 0, FILE_END, &filePtr);

   if (rc == 0) {
      printf("Set the file pointer at the end of the file.\n");
   }  else {
      printf("DosSetFilePtr failed with rc = %ld\n", rc);
   }

   /* Write a string to the file                                     */

   strcpy(buffer,"It is good that war is so terrible - we
      should grow too fond of it.\r\n");
   rc = DosWrite(FileHandle, buffer, strlen(buffer),
      &bytesWritten);

   if (rc == 0) {
      printf("Wrote %d bytes to the file.\n",bytesWritten);
   }  else {
      printf("DosWrite failed with rc = %ld.\n", rc);
   }

   rc = DosClose(FileHandle);

   if (rc == 0) {
      printf("Closed file %s.\n",filename);
   }  else {
      printf("DosClose failed with rc = %ld.\n", rc);
   }

   return;
}
```

Figure 5-1. *Program that opens, writes to, and closes a file (PROG5F1.C).*

```
Opened file FILECODE.TXT.
Set the file pointer at the end of the file.
Wrote 69 bytes to the file.
Closed file FILECODE.TXT.
```

Figure 5-2. *Output from the program in Figure 5-1 (PROG5F1.C).*

Reading from and Printing a File

The program in Figure 5-3 opens a file, reads from the file, and then prints and closes the file. This program introduces the DosRead API, which reads (copies) the number of bytes you specify from the file to a buffer. The syntax of DosRead follows.

DosRead

```
DosRead(
        HFILE    FileHandle,
        PVOID    BufferArea,
        ULONG    ulBufferLength,
        PULONG   BytesRead);
```

Parameters:

- ❑ FileHandle (HFILE) input—Indicates the name of a file handle obtained from a previous call to DosOpen.
- ❑ BufferArea (PVOID) input—Indicates the address of the buffer that will receive the bytes read from the file.
- ❑ ulBufferLength (ULONG) input—Specifies the number of bytes to be read by indicating the size of BufferArea.
- ❑ BytesRead (PULONG)—Specifies the address of the variable that will hold the number of bytes being read.

```
/******************************************************************/
/*   This program opens the file FILECODE.TXT, which was          */
/*   created by PROG5F1.C in Figure 5-1.  It sets                 */
/*   the file pointer to the beginning of the file,               */
/*   reads and prints the contents of the file,                   */
/*   and closes the file.  This is PROG5C3.C.                     */
/******************************************************************/
```

```
#define INCL_DOSFILEMGR                    /* File system values      */
#include <os2.h>
#include <stdio.h>

VOID main (USHORT argc, PCHAR argv[])
{
   HFILE   FileHandle;                    /* File handle to create   */
   ULONG   Action;                        /* Results of action       */
   APIRET  rc;                            /* Return code             */
   UCHAR   filename[20];
   ULONG   bytesWritten, bytesRead;
   ULONG   filePtr;
   BYTE    buffer[200];

   strcpy(filename,"FILECODE.TXT");
   rc = DosOpen(filename,                 /* Name of file to create  */
      &FileHandle,                        /* Address of file handle  */
      &Action,                            /* Pointer to action       */
      0,
      FILE_NORMAL,                        /* Attribute of new file   */
      FILE_OPEN,
      OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, NULL);

   if (rc == 0) {
      printf("Opened file %s.\n", filename);
   } else {
      printf("DosOpen failed with rc = %ld.\n", rc);
      return;
   }

   /* Not really necessary, since initially opening the file
      will set the file pointer to the beginning.               */

   rc = DosSetFilePtr(FileHandle, 0, FILE_BEGIN, &filePtr);

   if (rc == 0) {
      printf("Set the file pointer to the beginning of the
         file.\n");
   } else {
      printf("DosSetFilePtr failed with rc = %ld\n", rc);
   }

   rc = DosRead(FileHandle, buffer, sizeof(buffer),
      &bytesRead);

   if (rc == 0) {
      printf("Read %d bytes from the file.\n",bytesRead);
      printf("Contents:\n%s\n", (CHAR *) buffer);
   } else {
      printf("DosRead failed with rc = %ld.\n", rc);
   }
   rc = DosClose(FileHandle);

   if (rc == 0) {
      printf("Closed file %s.\n",filename);
   } else {
      printf("DosClose failed with rc = %ld.\n", rc);
   }
   return;
}
```

Figure 5-3. *Program that opens, reads, prints, and closes a file (PROG5F3.C)*

```
Opened file FILECODE.TXT.
Set the file pointer to the beginning of the file.
Read 69 bytes from the file.
Contents:
It is good that war is so terrible - we should grow too
  fond of it.

Closed file FILECODE.TXT.
```

Figure 5-4. *Output from the program in Figure 5-3 (PROG5F3C).*

Locking and Writing to a File

More than one application can open and write to a file at the same time in OS/2 2.1. To prevent another application from overwriting your work, you can lock a section (range) temporarily with the DosSetFileLock API. This API denies access to any other process that tries to write to that section. DosSetFileLock can also unlock the file when your operation is finished.

If the length of a section to be locked goes beyond the end of the file, the API works without an error. However, sections to be locked can't overlap, nor can a larger section encompass a smaller, previously locked section.

An ERROR_LOCK_VIOLATION will be returned on any attempt to lock a section or part of a section that is already locked. You'll have to clear up the conflicting locks before issuing new locks on the same section.

A file can have more than one lock, but the locks can't overlap. Figure 5-5 on a later page illustrates two successful locks, Lock A and Lock B, and one unsuccessful lock, Lock C.

Note that a duplicate file handle in the same process as the original file handle can access any locked sections that the original file handle is currently using. To prevent problems with a duplicate file handle overwriting the work of the original file handle, you can use DosExecPgm to create a child process that inherits the parent's file handle and passes it to a child if the child can be trusted not to destroy the parent's work.

The program in Figure 5-6 introduces the DosSetFileLocks API. The syntax of the DosSetFileLocks API follows.

DosSetFileLocks

```
DosSetFileLocks(
                    HFILE        FileHandle,
                    PFILELOCK    pUnLockRange,
                    PFILELOCK    pLockRange,
                    ULONG        ulTimeOut
                    ULONG        ulFlags);
```

Parameters:

- ❑ FileHandle (HFILE) input—Specifies the name of a file handle previously returned from a call to DosOpen.

- ❑ pUnLockRange (PFILELOCK) input—Indicates the address of the structure that contains the length of the range you want to unlock along with the offset to the beginning of the file. This structure contains two fields:

  FileOffset (LONG) input—Contains the offset to the beginning of the file; shows where to start the unlock action.

  RangeLength (LONG) input—Contains the length (size) of the area to be unlocked; 0 means don't unlock.

- ❑ pLockRange (PFILELOCK) input—Indicates the address of the structure that contains the length of the range you want to lock along with the offset to the beginning of the file. This structure contains two fields:

  FileOffset (LONG) input—Contains the offset to the beginning of the file; shows where to start the lock action.

  RangeLength (LONG) input—Contains the length (size) of the area to be locked; 0 means don't lock.

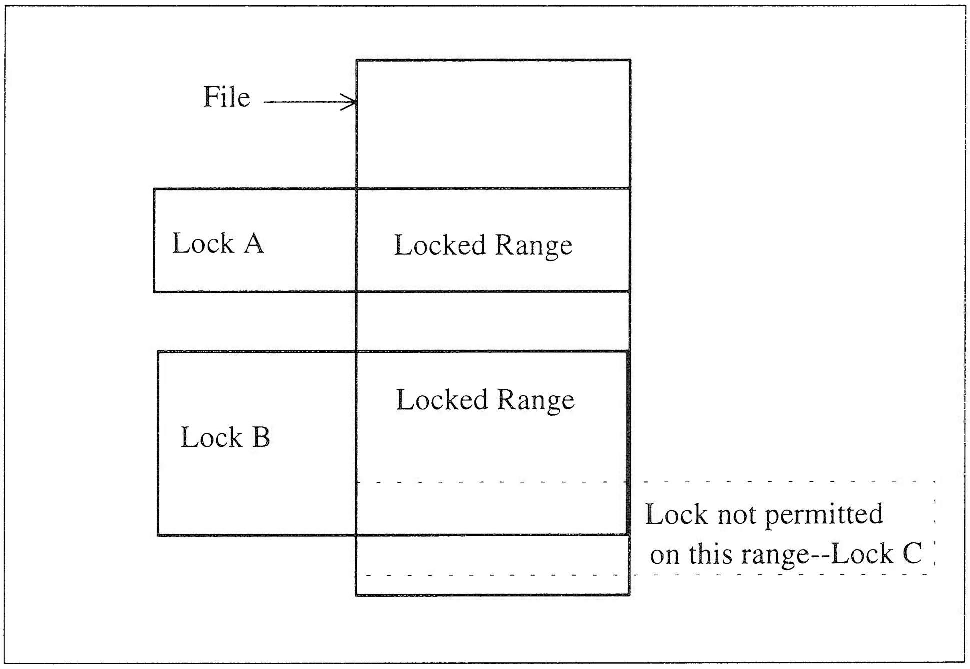


Figure 5-5. *Locked ranges cannot overlap.*

Note: Coordinate the values of the pLockRange and pUnLockRange structures. When you're locking a range of a file, set pUnLockRange to 0; when you're unlocking, set pLockRange to 0.

- ❑ ulTimeOut (ULONG) input—Indicates the maximum number of milliseconds the process will wait for the requested lock actions to be carried out.

- ❑ ulFlags (ULONG) input—Contains one of the following actions:

Bit	**Type**	**Values**	**Description**
0	Share	0,1	If set to the default (0), this bit denies access to the range to other processes; the current process has exclusive rights to the range. If set to 1, this bit permits read-only access to the range to the current process and other processes.
1	Atomic	0,1	Lets you do both operations (unlock and lock a file range) in the same call if three conditions are met.

Bit	Type	Values	Description
			Atomic bit=1
			Unlock range=lock range
			Process shares access but now requests exclusivity, or process has exclusive access but now requests sharing. Set this bit to force unlocking and locking in two calls.
2-32			Reserved.

The program in Figure 5-6 shows the DosSetFileLocks API in operation.

```
/*******************************************************************/
/*   PROG5F6.C.  DosSetFileLocks                                   */
/*   This program opens a file (creating it if it doesn't          */
/*   exist), locks the file, sets the file pointer to the          */
/*   end of the file, writes a text string, and closes the         */
/*   file.                                                         */
/*******************************************************************/


#define INCL_DOSFILEMGR                    /* File system values     */
#include <os2.h>
#include <stdio.h>

VOID main (USHORT argc, PCHAR argv[])
{
   HFILE    FileHandle;                    /* File handle to create  */
   ULONG    Action;                        /* Results of action      */
   APIRET   rc;                            /* Return code            */
   UCHAR    x[20], filename[20];
   ULONG    bytesWritten;
   ULONG    filePtr;
   ULONG    timeout = 3000;
   FILELOCK area;

   strcpy(filename,"FILECODE.TXT");
   rc = DosOpen(filename,                 /* Name of file to create  */
      &FileHandle,                        /* Address of file handle  */
      &Action,                            /* Pointer to action       */
      0,                                  /* Size of new file        */
      FILE_NORMAL,                        /* Attribute of new file   */
      FILE_OPEN | OPEN_ACTION_CREATE_IF_NEW,
      OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE, NULL);

   if (rc == 0) {
      printf("Opened file %s.\n", filename);
   } else {
      printf("DosOpen failed with rc = %ld.\n", rc);
```

```
    return;
 }


/*   Lock a region of the file so other processes can't          */
/*   get to it.                                                    */


area.lOffset = 0;             /* Where to begin the locked range. */
area.lRange = 20;             /* Range to lock out.               */

rc = DosSetFileLocks(FileHandle,                /* File handle      */
       0L,                                      /* Unlock range (0) */
       (PFILELOCK) &area,                       /* Lock range       */
       timeout,                                 /* Lock timeout     */
       0);                                      /* Share request    */

if (rc == 0) {
    printf("DosSetFileLocks succeeded in locking
       bytes %d to %d.\n",area.lOffset,area.lOffset+area.lRange);
 } else {
    printf("DosSetFileLocks failed with rc = %ld,\n", rc);
}

 rc = DosSetFilePtr(FileHandle, 0, FILE_END, &filePtr);

 if (rc == 0) {
    printf("Set the file pointer at the end of the file.\n");
 } else {
    printf("DosSetFilePtr failed with rc = %ld\n", rc);
 }

 strcpy(x,"--General Robert E. Lee\r\n");
 rc = DosWrite(FileHandle, x, strlen(x), &bytesWritten);

 if (rc == 0) {
    printf("Wrote %d bytes to the file.\n",bytesWritten);
 } else {
    printf("DosWrite failed with rc = %ld.\n", rc);
 }

 rc = DosSetFilePtr(FileHandle, 0, FILE_BEGIN, &filePtr);

 if (rc == 0) {
    printf("Set the file pointer at the beginning of the file.\n");
 } else {
    printf("DosSetFilePtr failed with rc = %ld\n", rc);
 }

 rc = DosClose(FileHandle);

 if (rc == 0) {
    printf("Closed file %s.\n",filename);
 } else {
      printf("DosClose failed with rc = %ld.\n", rc);
   }

   return;
}
```

Figure 5-6. *Program that locks a file.*

```
Opened file FILECODE.TXT.
DosSetFileLocks succeeded in locking bytes 0 to 20.
Set the file pointer at the end of the file.
Wrote 25 bytes to the file.
Set the file pointer at the beginning of the file.
Closed file FILECODE.TXT.
```

Figure 5-7. *Output of the program in Figure 5-6 (PROG5F6.C).*

Note that the DosSetFileLocks API in Figure 5-6 locks the range of the file only during the write operation. The call to the DosClose API automaticallly releases any locks but does so in an unpredictable manner.

APIs ASSOCIATED WITH FILES AS A WHOLE

APIs for copying, deleting, and moving files are discussed in this section. The program in Figure 5-8 then illustrates the use of DosCopy, DosDelete, and DosMove.

Copying a File

Use the DosCopy API to place a copy of an individual file or a whole subdirectory of files at another location, which can be on a different drive.

DosCopy does not support the metacharacters, asterisk (*) and question mark (?), in the file or subdirectory name.

In addition to the file itself, DosCopy also copies the attributes of the file, such as the time and date the file was created and the size of the file. If the file has extended attributes, these are also copied to the new location.

The syntax of the DosCopy API will give you enough information to use this API in your applications.

DosCopy

```
        DosCopy(
                PSZ    SourceName
                PSZ    TargetName
                ULONG  ulOpMode);
```

Parameters:

- ❑ SourceName (PSZ) input—Indicates the address of the file to be copied; can also be used to indicate the address of a subdirectory or character device.
- ❑ TargetName (PSZ) input—Indicates the address to which the file will be copied; can also be used to indicate the address of a subdirectory or character device.
- ❑ OpMode (ULONG) input—Indicates how to handle the copy operation in various situations:

Bit	Name	Description
0	DCOPY_EXISTING	If set to 0, do not copy if the target file name already exists; if set to 1, copy anyway (overwrite).
1	DCPY_APPEND	If set to 0, replace the target file with the source file; if set to 1, append the source file to the end of the target file.
3-31		Reserved; must be 0.

Deleting a File

Use the DosDelete API to remove all types of files except those with a file attribute of FILE_READ_ONLY from a directory. To delete a read-only file, first change the file attribute to FILE_NORMAL and then delete it.

DosDelete does not support metacharacters, the asterisk (*) and the question mark (?), in the file or subdirectory name.

The syntax of the DosDelete API will give you enough information to use the API in your applications.

DosDelete

```
          DosDelete(
                    PSZ  FileName);
```

Parameter:

- ❑ FileName (PSZ) input—Indicates the address of the file to delete.

Moving a File

TheDosMove API removes a file from its original location and stores it in another location. If a file has a long file name, between 9 and 255 characters, you can move it to a disk that does not support long file names. The syntax of the DosMove API will give you enough information for you to use the API in your applications

DosMove

```
          DosMove(
                  PSZ   OldPathName,
                  PSZ   NewPathName);
```

Parameters:

- ❑ OldPathName (PSZ) input—Indicates the address of the existing pathname of the file to be moved.

- ❑ NewPathName)PSZ) input—Indicates the address of the new pathname of the file to be moved.

The program in Figure 5-8 illustrates DosCopy, DosDelete, and DosMove. It copies and replaces the CONFIG.SYS file. Run it only from the C drive.

```
/*  Prog5F8.C.  Illustration of DosCopy, DosMove, and DosDelete.    */

#include <stdio.h>
#define INCL_DOSFILEMGR
#include <os2.h>

INT main(VOID)
{
   APIRET rc;
   rc = DosCopy("C:\\CONFIG.SYS", ".\\CONFIG.BAK",
                DCPY_EXISTING | DCPY_FAILEAS);

   if (!rc) {
      DosDelete("C:\\CONFIG.SYS");

      rc = DosMove(".\\CONFIG.BAK", "C:\\CONFIG.SYS");
      if (!rc) {
         DosDelete(".\\CONFIG.BAK");
      } else {
         printf("Unable to move file.  RC = %d\n", rc);
         return(1L);
      }
   } else {
      printf("Unable to copy CONFIG.SYS to CONFIG.BAK");
      printf(" rc = %d\n", rc);
      return(1L);
   }
   printf("CONFIG.SYS copied and replaced!\n");

   return(0L);
}
```

Figure 5-8. *Program illustrating DosCopy, DosMove, and DosDelete (PROG5F8.C).*

Searching for a File

APIs that search for files in the current directory include DosFindFirst and DosFindNext. DosFindClose then closes the directory handle when the search is finished. The program in Figure 5-9 then illustgrates these APIs in action. The syntax of DosFindFirst follows.

DosFindFirst

```
DosFindFirst(
             PSZ      FileName,
             PHDIR    pDirHandle,
             ULONG    ulAttribute,
             PVOID    ResultBuf,
             ULONG    ulResultBufLen,
             PULONG   SearchCount,
             ULONG    uFileInfoLevel);
```

Parameters:

- ❑ FileName (PSZ) input—Indicates the address of the file to be searched for and found.

- pDirHandle (PHDIR) input or output—Indicates the address of the handle to use for output in this search:

  HDIR_SYSTEM specifies to use the system handle for standard output.

  HDIR_CREATE specifies to allocate a new handle for output.

- ulAttribute (ULONG) input—Indicates a value to determine the file object to search for:

Bit	**Description**
0	FILE_READONLY
1	FILE_HIDDEN
2	FILE_SYSTEM
3	Reserved; must be 0.
4	FILE_DIRECTORY
5	FILE_ARCHIVED
6-7	Reserved; must be 0.
8	MUST_HAVE_READONLY
9	MUST_HAVE_HIDDEN
10	MUST_HAVE_SYSTEM
11	Reserved; must be 0.
12	MUST_HAVE_DIRECTORY
13	MUST_HAVE_ARCHIVED

- ResultBuf (PVOID) input or output—On input, this indicates the address of directory structures to search for levels of information (Level 1, Level 2, or Level 3); on output, this indicates a structure containing information on levels.

- ulResultBufLen (ULONG) input—Indicates the length of the result buffer in bytes.

- SearchCount (PULONG) input or output—On input, this indicates the hits requested; on output, this contains the number of hits found.

- uFileInfoLevel (ULONG) input—Indicates the level of information requested.

The syntax of the DosFindNext API uses a subset of the DosFindFirst API parameters (DirHandle, ResultBuf, ResultBufLen, and SearchCount). The syntax of the DosFindClose API uses an even smaller subset of the DosFindFirst API parameters (DirHandle).

The DosFindFirst API locates the first instance of a file name that matches the pattern of the search argument. The DosFindNext API locates the next instance; subsequent calls will continue until all the matching names are found. Then the message ERROR_NO_MORE_FILES is returned to the application. Both search APIs use a handle (DirHandle) to keep track of the files that have been found. Use the directory handle returned by DosFindFirst in calls to DosFindNext because this variable identifies the name of the file to search for and the current position in the directory.

If you want to receive additional information from the search APIs beyond a count of the matching file names found, you can use a structure to hold information about the file, such as its name, the date and time it was created, the length of data in the file, the size of the file, and the date it was last accessed. This type of information is called Level 1 information.

The program in Figure 5-9 illustrates the use of DosFindFirst, DosFindNext, and DosFindClose to search for multiple files.

```
/*  Prog5F9.C.  Program illustrating searching for multiple files  */
/*             using DosFindFirst, DosFindNext, and DosClose.    */

#include <stdio.h>
#define INCL_DOSFILEMGR
#include <os2.h>

INT main( INT argc, CHAR *argv[], CHAR *envp[] )
{
   APIRET       rc;
   HDIR         handleFileFirst = 0x0001;
   ULONG        count = 1, numfiles = 0;
   FILEFINDBUF3 findbuf;

   if (argc < 2) {
      printf("Please pass file search string\n");
      return(1L);
   }


   rc = DosFindFirst(argv[1], &handleFileFirst, 0,
                     (PVOID)&findbuf, sizeof(findbuf),
                     &count, FIL_STANDARD);
   while (!rc) {
      printf("Matching file %s found\n", findbuf.achName);
      numfiles++;
      rc = DosFindNext(handleFileFirst, (PVOID)&findbuf,
                       sizeof(findbuf), &count);
   }
   DosFindClose(handleFileFirst);
   printf("There were %d matching files.\n", numfiles);
   return(0L);
}
```

Figure 5-9. *Program illustrating searching for multiple files using DosFindFirst, DosFindNext, and DosFindClose.*

DosEditName lets an application transform a file name into another name by using a template that contains metacharacters.

- ❑ An asterisk (*) in the template indicates to copy the characters in the existing name until it finds a character that matches the one that comes after the asterisk.
- ❑ A question mark (?) in the template indicates to copy one character.
- ❑ If the character is a period, however, this indicates to move to the next period in the existing name and skip all the characters in between the current position and the period.

The syntax of the DosEditName API will give you enough information to use the API in your applications.

DosEditName

```
        DosEditName(
                    ULONG   ulEditLevel,
                    PSZ     SourceString,
                    PSZ     EditString
                    PBYTE   bTargetBuf
                    ULONG   TargetBufLen);
```

Parameters:

- ❑ ulEditLevel (ULONG) input—Indicates the type of editing to use in changing the source.
- ❑ SourceString (PSZ) input—Indicates the address of the string to transform.
- ❑ EditString (PSZ) input—Indicates the address of the string to use as a template against which to transform the source string.

- ❑ bTargetBuf (PBYTE) output—Indicates the address of the buffer containing the transformed string.
- ❑ TargetBufLen (ULONG) input—Indicates the length of the transformed string in bytes.

MISCELLANEOUS APIs ASSOCIATED WITH FILE LOCATIONS

Locations associated with files include paths, disks and drives, volumes or partitions, and directories. File system APIs for these locations include such operations as searching for and retrieving information, setting information, and creating and deleting directories.

Paths

Perhaps the most common path operation is for an application to search a path for a file, using DosSearchPath. You can specify that the search start with the current directory before the path is searched.

When you specify the path, you can either supply a string, in which the items are separated by semicolons, or you can specify the name of an environment variable, such as DPATH. The program in Figure 5-10 illustrates searching for multiple files. The syntax of the DosSearchPath API follows.

DosSearchPath

```
DosSearchPath(
                ULONG  ulControl,
                PSZ    PathRef,
                PSZ    FileName,
                PBYTE  bResultBuffer,
                ULONG  ulResultBufferLen);
```

Parameters:

- ❑ ulControl (ULONG) input—Indicates where and how DosSearchPath will operate, as controlled by word bit vector settings.

Bit	**Name**	**Description**
0	SEARCH_CUR_DIRECTORY	A setting of 0 causes only the current directory to be searched if the current directory is in the search path; a setting of 1 causes the current working directory to be searched before any directories in the search path.
1	SEARCH_ENVIRONMENT	Bit 1 controls the PathRef parameter. A setting of 0 makes PathRef point to the actual search path anywhere in the address space of the process; a setting of 1 makes PathRef point, in the processes's environment, to an environment variable that contains the search path.
2	SEARCH_IGNORENETERRS	Controls whether to pay attention to network errors. A setting of 0 causes a search to end if a network error is received; a setting of 1 causes the search to continue despite network errors received.
3		Reserved; must be 0.

- ❑ PathRef (PSZ) input—Indicates the address of the path, depending on the bit setting in the Control parameter.
- ❑ FileName (PSZ) input—Indicates the address of the file to be searched for.
- ❑ bResultBuffer (PBYTE) output—Contains the result of the search: the file name if found or unpredictable data if not found.
- ❑ ulResultBufferLen (ULONG) input—Specifies the length of the result buffer in bytes.

Figure 5-10 shows DosSearchPath in action.

```
/*    PROG5F10.C.  Program illustrating DosSearchPath.                    */

#include <stdio.h>

#define INCL_DOSMISC
#include <os2.h>

VOID SearchPath(CHAR * file, CHAR * path)
{
   APIRET rc;
   CHAR   buffer[200];

   rc = DosSearchPath(SEARCH_ENVIRONMENT |
                      SEARCH_IGNORENETERRS,
                      path, file, buffer, sizeof(buffer));
   if (!rc) {
      printf("Your file was found within your %s!\n", path);
   }
}

INT main( INT argc, CHAR *argv[], CHAR *envp[] )
{

   if (argc < 2) {
      printf("Please pass a file name\n");
      return(1L);
   }

   SearchPath(argv[1], "DPATH");
   SearchPath(argv[1], "PATH");
   SearchPath(argv[1], "HELP");

   return(0L);
}
```

Figure 5-10. *Program illustrating searching for a file in a file path using DosSearchPath.*

An application can also call a set of APIs to gather or change information associated with the path to a file or subdirectory.

DosQueryPathInfo and DosSetPathInfo obtain or set Level 1 file information, which includes the size of the file, the date and time it was created, the date it was last accessed, and the date it was last written to.

The DosQuerySysInfo API lets an application find out the maximum length for a path name that is supported by the current file system.

Volumes, Disks, and Drives

When a process starts running, it uses the same volume, disk, and drive of the calling application. The process may need to know what these values are and may sometimes need to change them.

The major APIs that manage these interactions between files and volumes, disks, and drives include the following:

- ❑ DosQueryCurrentDisk lets an application retrieve the name of the default drive.
- ❑ DosSetCurrentDisk lets an application set the name of the default drive.
- ❑ DosSetDefaultDisk also lets an application set the name of the default drive.
- ❑ DosPhysicalDisk lets an application retrieve information about a disk that can be partitioned.
- ❑ DosSetFSInfo and DosQueryFSInfo, which is discussed in Chapter 6, let an application set and retrieve Level 1 drive information.

Directories

When a process starts running, it uses the same directory as the calling application. The process may need to know the name of this directory and may sometimes need to change directories.

The APIs that handle these interactions between files and directories include the following:

- ❑ DosCreateDir lets an application create a new subdirectory in one of the following locations:

  At the end of the current directory, if no path name is specified.

  At the end of the drive and directory path name specified. If any items in the path don't exist, the subdirectory is not created.

- DosDeleteDir lets an application delete a directory at the end of the specified path. However, an application can't delete the current directory or any directory containing hidden files or subdirectories.

- DosQueryCurrentDir lets an application retrieve the full path name of the current directory on the specified drive. This full path name can begin with any character except the back slash (\). It ends with a byte containing the null character (00H).

- DosSetCurrentDir lets an application change to a different directory as specified in the path name.

File Handles and File Pointers

The following APIs provide an application with the means of managing file handles and file pointers:

- DosSetFilePtr (See Figure 5-1.)

- DosQueryFHState

- DosSetFHState

- DosSetMaxFH

Every file on a hard or floppy disk has a file pointer. A file pointer is a position marker in an open file that indicates one of the following, depending on whether a read or a write operation is in progress:

- The next byte that will be read

- The location to which the next byte will be written

A file pointer is automatically generated when a file is opened. The file pointer is placed at the beginning of the file and is advanced automatically with each read or write. However, an application can also move the file pointer by means of the DosSetFilePtr API. This API moves the file pointer a specified number of places (an offset) from a given position (the beginning of the file, the end of the file, or the current position in the file). Note that this API can be used only for files on disk but not for files on devices.

OS/2 2.1 provides APIs that let you query and reset the state of a file handle. DosQueryFHState returns the state bits set by DosOpen. DosSetFHState returns a file handle with the state bits changed as specified. This file handle is then used for all subsequent input and output during the current process.

Table 5-1 shows the state bits that DosSetFHState can set. Each item can be set in one of two ways except the last one. Cacheing is determined by the file system.

Name	**Meaning**
Inheritance	A child process created with DosExecPgm inherits the handle, or the handle is set as private to the calling process.
Write-Through	Input and output for writes is not guaranteed complete and error free or is guaranteed only before the write operation returns.
Fail_Errors	Either the system critical error handler or the application is set to report media errors.
DASD	Only systems programs should set the DASD flag, not regular applications, because it provides direct access to a disk or partition independent of the file system being used. When set, a file name parameter applies to a drive specification for an entire hard disk or diskette drive. The handle returned by DosOpen considers the logical volume as a single file. If a systems program sets this flag, it should also call a DosCevIOCtl Category 8, Function 0, to prevent other processes from accessing the logical volume.
Cache	The file system you are using either caches or does not cache data for input and output operations on the file. You can retrieve this information, but you can't change it.

Table 5-1. *Information that DosSetFHState can change.*

The DosSetFHState API is very useful in two situations. In the first example, you need to ensure that data is written in a specified order. Therefore, the application must do the writes in separate synchronous operations. Use the DosSetFHState API to set the

Write-through flag. If you did any asynchronous writes earliler, these won't be affected. This data can safely stay in the buffers.

In the second example, you're concerned that an application won't be able to handle a critical error. You can use DosSetFHState to set Fail_Errors so that the operating system, not the application, will handle the error.

You'll have to make the I/O error happen again by reissuing the offending function. Then pass control to the operating system. If you're doing asynchronous I/O, there's no means of determining when the results of the offending function will be returned to the application.

The Type of a File, Pipe, or Device Handle

To find out whether a handle belongs to a file, a pipe, or a device, an application can call the DosQueryHType API

MISCELLANEOUS FILE MANAGEMENT APIs

This section explains additional file management APIs that you may find useful from time to time. These are usually used in connection with DosWrite.

If you are writing critically important data to a disk, you may want to use the following APIs to guarantee the accuracy of the data being written:

- DosQueryVerify finds out whether the verification switch is on. If it is, the data that was written is verified to ensure that it was recorded correctly.

- DosSetVerify turns the verification switch on or off. Note that OS/2 2.1 does not support the 16-bit API DosWriteAsync.

  To prevent an appplication from being blocked during a write, create a separate thread and then post a semaphore when the write is finished.

SUMMARY OF FILE MANAGEMENT DATA STRUCTURES AND APIs

Table 5-2 lists the data structures used in file management.

Name	**Description**
DENA1	Contains Level 1 information returned by DosEnumAttribute.
FDATE	Provides a substructure for the FILEFINDBUF and FILESTATUS structures; contains information on dates, such as date created and date last changed.
FILEFINDBUF	Returns information about files when used by DosFindFirst or DosFindNext; FILEFINDBUF2, FILEFINDBUF3, and FILEFINDBUF4 can be used for variations of this data structure.
FILELOCK	Indicates the range to lock in the specified file when used with DosSetFileLocks.
FILESTATUS	Contains file date, time, and size when used with query or set information APIs.
TIME	Provides a substructure for the FILEFINDBUF and FILESTATUS structures; contains information on times, such as time created and time last changed.
VOLUMELABEL	Provides a substructure for the FSINFO file system data structure; contains the label of the volume.

Table 5-2. *File Management Data Structures.*

Table 5-3 provides a summary of file APIs.

Name	**Description**
DosClose	Closes a file handle.
DosCopy	Copies a file or subdirectory.
DosDelete	Deletes a file.
DosEditName	Changes a file name.
DosMove	Moves a file or directory.

Name	**Description**
DosOpen	Gets a handle to a file, pipe, or device.
DosRead	Reads from a file, pipe, or device.
DosSetFileInfo	Sets certain information for an open file.
DosSetFileLocks	Locks or unlocks a range in a file.
DosSetFilePtr	Sets the file pointer to a new position.
DosSetPathInfo	Sets certain information for a file or subdirectory.
DosSetVerify	Turns on write verification.
DosWrite	Writes data to a file, pipe, or device.

Table 5-3. *File APIs.*

Table 5-4 provides a summary of file query APIs.

Name	**Description**
DosEnumAttribute	Gets the name and size of the extended attributes of a file object.
DosQueryFileInfo	Gets information for an open file.
DosQueryPathInfo	Gets information for a file or subdirectory.
DosQuerySysInfo	Gets the values of system variables.
DosQueryVerify	Gets information on whether write verification is turned on.

Table 5-4. *File Query APIs.*

Table 5-5 provides a summary of file handle APIs:

Name	**Description**
DosDupHandle	Makes a duplicate of a file handle.
DosQueryFHState	Gets the state of the file handle.
DosQueryHType	Gets the type of the handle.
DosSetFHState	Sets the state of the file handle.

Name	**Description**
DosSetMaxFH	Sets the maximum number of available file handles.

Table 5-5. *File Handle APIs.*

Table 5-6 provides a summary of file search APIs:

Name	**Description**
DosFindClose	Stops a search for matching file objects.
DosFindFirst	Starts a search for matching file objects.
DosFindNext	Continues a search for matching file objects.

Table 5-6. *Directory Search APIs.*

Table 5-7 provides a summary of directory and disk APIs:

Name	**Description**
DosCreateDir	Creates a subdirectory.
DosDeleteDir	Erases an empty subdirectory.
DosQueryCurrentDir	Gets the name of the current directory.
DosQueryCurrentDisk	Gets the name of the current drive.
DosSetCurrentDir	Sets the current directory.
DosSetDefaultDisk	Sets the default drive.

Table 5-7. *Directory and Disk APIs.*

Table 5-8 provides a summary of environment and search path APIs:

Name	**Description**
DosScanEnv	Scans environment variables.
DosSearchPath	Searches a specified path.

Table 5-8. *Environment and Search Path APIs.*

CHAPTER 6

Files and File Systems

"Our little systems have their day." —*Tennyson*

INTRODUCTION

This chapter discusses topics relating to files other than managing and using files. They include:

- ❑ Files
- ❑ File handles
- ❑ File systems
- ❑ File names and attributes

FILES

An application program frequently has to access and manipulate files because a file is the basic storage unit for data. Measured in bytes, a file resides on a storage device, such as a hard disk, a floppy disk, or CD-ROM. In OS/2 2.1, files are created and organized in one of two types of file systems provided by the operating system. The two types are the File Allocation Table (FAT) and the Installable File System (IFS).

A file looks like a steady stream of bytes to the application that is performing an I/O operation, such as reading the contents of the file.

An application can retrieve the following information about a file with the DosQueryFileInfo API:

- ❏ The date and time the file was created.
- ❏ The date and time the file was last accessed.
- ❏ The date and time the file was last written to.
- ❏ The attributes of the file.
- ❏ The number of bytes in the file.
- ❏ The number of sectors on the disk required to store the file.
- ❏ For large files, the number of clusters on the disk required to store the file.

FILE HANDLES

In OS/2 2.1, a file handle is a unique 32-bit integer that associates an open file with an application so that the application can use the file handle instead of the name, which can include drive\path\filename.extension.

The file handle is assigned to a file when an application creates it or opens an existing file with the DosOpen API. A file handle has no meaning other than as an identifier. A useful programming technique is to store the file handle as a global variable.

The maximum number of file handles an application can have open is 20, unless you increase the maximum with the DosSetMaxFH API. This API also saves all file handles that are currently open. An application inherits open file handles from the process that starts it. Every application also has three standard file handles that are opened automatically, as Table 6-1 shows:

File Handle	Number	Description
Standard Input	0	Used with DosRead to read characters from the keyboard.
Standard Output	1	Used with DosWrite to write characters to the screen.

File Handle	Number	Description
Standard Error	2	Used to display error messages to the screen.

Table 6-1. *Standard file handles.*

The program in Figure 6-1 shows how to open the D:\CONFIG.SYS file, retrieve Level 1 information, print the size of the file, and then close the file. It uses the DosQueryFileInfo API.

DosQueryFileInfo

```
DosQueryFileInfo(
                  HFILE   FileHandle,
                  ULONG   uFileInfoLevel,
                  PVOID   FileInfoBuf,
                  ULONG   uFileInfoBufSize);
```

Parameters:

❑ FileHandle (HFILE) input—Indicates the handle of the file to get information about.

❑ uFileInfoLevel (ULONG) input—Indicates the level of information to be returned:

Level	Name	Description
1	FIL_STANDARD	Returns Level 1 information in the FILESTATUS3 structure.
2	FIL_QUERYEASIZE	Returns Level 2 information in the FILESTATUS4 structure.
3	FIL_QUERYEASFROMLIST	Returns Level 3 information in the EAOP2 structure.

❑ FileInfoBuf (PVOID) output—Indicates the address to which the requested level of information is returned.

❑ uFileInfoBufSize (ULONG) input—Indicates the length of FileInfoBuf in bytes.

The program in Figure 6-1 shows DosQueryFileInfo in action.

```
/*   PROG6A.C.  Illustration of DosQueryFileInfo                    */

#define INCL_DOSFILEMGR                /* File system values         */
#include <os2.h>
#include <stdio.h>

VOID main (USHORT argc, PCHAR argv[])
{
   HFILE      FileHandle;                 /*  File handle            */
   ULONG      FileInfoLev;                /*  Level of info to return*/
   FILESTATUS FileInfoBuffer;             /*  Buffer for information */
   ULONG      FileInfoBufferSize;         /*  Size of the buffer     */
   ULONG      ActionTaken;
   APIRET     rc;                         /*  Return code            */
   UCHAR      FileName[20];

   FileInfoLev = 1;                       /* Return Level 1 info     */
   FileInfoBufferSize = sizeof(FILESTATUS);          /* Set size     */
   strcpy(FileName,"D:\\CONFIG.SYS");

   rc = DosOpen(FileName,
                &FileHandle,
                &ActionTaken,
                0,                                      /* File Size */
                FILE_NORMAL,
                FILE_OPEN,
                OPEN_SHARE_DENYNONE | OPEN_ACCESS_READWRITE,
                0L );                     /* No extended attributes */

   if (rc == 0) {
      printf("DosOpen for %s executed successfully.\n", FileName);
   } else {
      printf("Error:  DosOpen for %s with rc = %ld\n", FileName,rc);
      return;
   }

 rc = DosQueryFileInfo(FileHandle,
                       FileInfoLev,
                       &FileInfoBuffer,
                       FileInfoBufferSize);
   if (rc == 0) {
      printf("DosQueryFileInfo for %s executed
              successfully.\n",FileName);
      printf("  File size is %d.\n",FileInfoBuffer.cbFile);
   } else {
      printf("Error:  DosQueryFileInfo for %s with rc =
             %ld\n",FileName,rc);
   }
   rc = DosClose(FileHandle);

   if (rc == 0) {
      printf("DosClose for %s executed
              successfully.\n",FileName);
   } else {
      printf("Error:  DosClose for %s with rc = %ld\n",
             FileName,rc);
   }
   return;
}
```

Figure 6-1. *DosQueryFileInfo* (PROG6A.C).

```
DosOpen for D:\CONFIG.SYS executed successfully.
DosQueryFileInfo for D:\CONFIG.SYS executed successfully.
File size is 4386.
DosClose for D:\CONFIG.SYS executed successfully.
```

Figure 6-2. *Output from the program in Figure 6-1* (PROG6A.C).

FILE SYSTEMS

File systems consist of hardware and software that stores or retrieves information to or from some type of storage device. Devices such as hard disks, floppy disks, and CD-ROM drives comprise the hardware part of a file system.

A hard disk can be divided into partitions and logical drives if needed. The software part of a file system includes drivers that control how the information is organized on the storage devices, such as by directories, subdirectories, and files.

The two types of file systems that OS/2 2.1 provides can be active at the same time but not in the same partition. For example, if you divide the hard disk into two partitions, you can use one file system in the first partition and another file system in the second partition. During installation of OS/2 2.1, you can partition the hard disk and select the file system for each drive.

Following are the two types of file systems:

- ❑ The *File Allocation Table* (FAT) type of file system can be used by applications written to DOS and to all versions of OS/2. It is supplied with the operating system and does not require separate installation. It is recommended that you use FAT for the partition on which the operating system resides.

- The *Installable File System* (IFS) type requires separate installation. Currently, the High Performance File System (HPFS) is an IFS that is supplied with OS/2 1.3 and later versions. It can be used by applications written to OS/2 1.3 and later but not by those written to DOS and earlier versions of OS/2. The length of the file name is one such incompatibility between the two types of file systems. Other IFS products, such as support for redirected installation of clients, are also available.

OS/2 2.1 applications can access files created under DOS and earlier versions of OS/2 because most of the common file system APIs do not depend on the type of file system in use. Both FAT and HPFS support the following:

- The naming conventions established under DOS
- The logical file and directory structure established under DOS
- The processing of files whose names include metacharacters (wild cards) such as the asterisk (*) and the question mark (?)
- More than one partition on the hard disk
- Access to more than one storage device for files
- Access to more than one type of storage device for files
- Connections to files on other computers in a network (known as remote file access or redirection)
- Extended attributes, which apply to both FAT and IFS files (see Chapter 7)
- A maximum file size of 2 gigabytes

The File Allocation Table (FAT)

Since the OS/2 FAT is based on the DOS FAT, it is already familiar to you as a tree-structured arrangement of directory names and file names. From the A: and B: drives for floppy disks or the C:, D:, and subsequent drives for hard disks and network or CD-ROM drives, you know how to create directories with names up to eight characters long and files with names up to eight characters long with optional three-character extensions. You are also knowledgeable about the C functions in DOS or the APIs in earlier levels of OS/2 that create, open, read, write, and close FAT files. You can carry this knowledge over to the HPFS as a base for learning about the new features of HPFS.

The High Performance File System (HPFS)

HPFS is designed to provide faster access to larger numbers of very large files than FAT does. It is recommended that you install HPFS in partitions other than the primary partition in which the operating system resides.

The software part of the HPFS contains device drivers, which perform the access to APIs, and dynamic link libraries (DLLs), which control how information is formatted on a device and how the information flows to and from the device. In addition, HPFS provides the following:

- ❑ Cache memory (2KB) supports fast access to directories, file system (HPFS) data structures, and data.
- ❑ Directory structures are allocated strategically.
- ❑ Files are allocated contiguous to each other.
- ❑ Extended attributes are supported. Extended attributes are discussed in Chapter 7.
- ❑ Long file names, up to 255 characters, are supported. The last character in the string must be the null terminator (\0). A path name containing the drive, directories, and file name, can be 260 characters, including the required null terminator.
- ❑ File names containing blank spaces are supported if the section of the name that contains a blank is surrounded with double quotation marks (for example, C:\"OS2 2.0 Desktop").
- ❑ I/O operations can use more than one thread, which can improve the performance and reliability of the operation.

- ❑ An application can write information to the hard disk as a background activity, which can make I/O operations more efficient. However, immediate write, which is done in the foreground, is still supported. For critically important write operations, you may want to use immediate write since background writes are lost if the computer loses power and goes down.

- ❑ You can start OS/2 2.1 from a partition formatted for HPFS or for FAT.

- ❑ HPFS supports partitions up to 512 gigabytes.

The CONFIG.SYS file is updated when you select HPFS during the installation of OS/2 2.1. A DEVICE= statement is added to include the device driver, HPFS.IFS. Also, an IFS= statement is added to include the two dynamic link libraries:

- ❑ STARTLW.DLL—Enables background writing, also known as *lazy writing*, to disk.

- ❑ UPHFS.DLL—Enables the HPFS utilities.

The driver and DLLs account for three of the four modules of the HPFS. The final module is CACHE.EXE, which is loaded onto the hard disk during installation. This module controls the 2KB of cache memory that is used to speed up data access. You can set the following parameters on a call to CACHE.EXE from the command line:

- ❑ MaxAge is a time limit for data to remain in the cache, regardless of I/O activity. A block of data that exceeds the time limit is queued for writing to the hard disk.

- ❑ DiskIdle is the amount of time to wait between foreground I/O requests before queuing certain blocks of data in the cache for writing. This parameter is not used with background, or lazy, writing.

- ❑ BufferIdle determines which blocks of data to write when DiskIdle is exceeded. All blocks of data that have been inactive for this amount of time are queued for writing to disk.

These parameters help to minimize the amount of data lost in power outages and help you manage the read and write operations efficiently.

Deciding Whether to Install and Use HPFS

If your applications will never need rapid access to large amounts of data and if you'll never need to use long file names, you should probably consider staying with the FAT file system.

The main advantage of HPFS is to improve performance when an application must open and close a large number of very large files as well as perform numerous read and write operations on those files.

If you are considering using HPFS, you should first determine whether you need to continue providing access to existing FAT files. If you don't, you can simply install OS/2 2.1 as the only operating system on your computer and select HPFS as your only file system. You can still run DOS or Windows programs under OS/2 2.1.

If you decide to use both the FAT and the HPFS file systems, you'll need to partition the hard disk. Figure 6-3 on the next page shows one of many possible schemes for managing the hard disk.

This scheme makes use of the Boot Manager, which displays a menu from which you can select the operating system that you want to use. The drive containing the Boot Manager is not assigned a drive name.

An alternate operating system that you may want to run is in the second primary partition in Figure 6-3. A version of DOS, an earlier version of OS/2, or AIX, which is a UNIX-based operating system, are examples of alternate operating systems.

OS/2 2.1 is in the third primary partition. OS/2 2.1 and the other operating system in the second primary partition, in effect, share the C: drive since they can't both be active at the same time.

The fourth partition is known as the extended partition. Unlike a primary partition which can't be subdivided into logical drives, the extended partition can be subdivided into as many logical drives as you need, from D: through Z: drives.

Figure 6-3 shows two logical drives, D: formatted for FAT and E: formatted for HPFS.

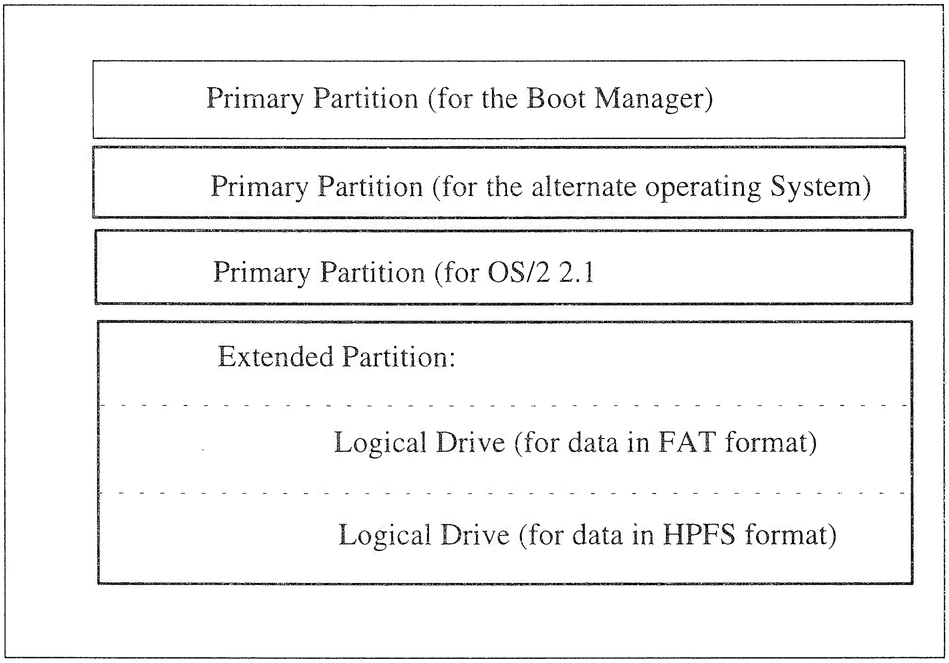

Primary Partition (for the Boot Manager)
Primary Partition (for the alternate operating System)
Primary Partition (for OS/2 2.1
Extended Partition:
Logical Drive (for data in FAT format)
Logical Drive (for data in HPFS format)

Figure 6-3. *Partitions on the hard disk.*

Applications, compilers, and data can be stored in logical drives, as well as in primary partitions. It's convenient to keep only the operating system in the primary partition so that installing a later version of the operating system is simplified.

Storage Devices

Each storage device and logical drive has a unique drive letter. By convention, the first floppy drive is the A: drive and the second, if it exists, is the B: drive. Similarly, the first hard disk is the C: drive and the second, if it exists, is the D: drive.

A maximum of four primary partitions is supported. However, the extended partition can contain logical drives up through the Z: drive.

Typically, the FDISK utility is offered to you during OS/2 2.1 installation so that you can partition the hard file if you want to. Also during installation, a file system is attached by formatting each partition or drive for the selected file system and by adding an IFS= statement to CONFIG.SYS that loads the file system driver each time you start the computer. HPFS, however, cannot be used for removable media, such as floppy drives. Figure 6-4 shows one of many possible ways to arrange storage devices and drives in a system with two hard disks.

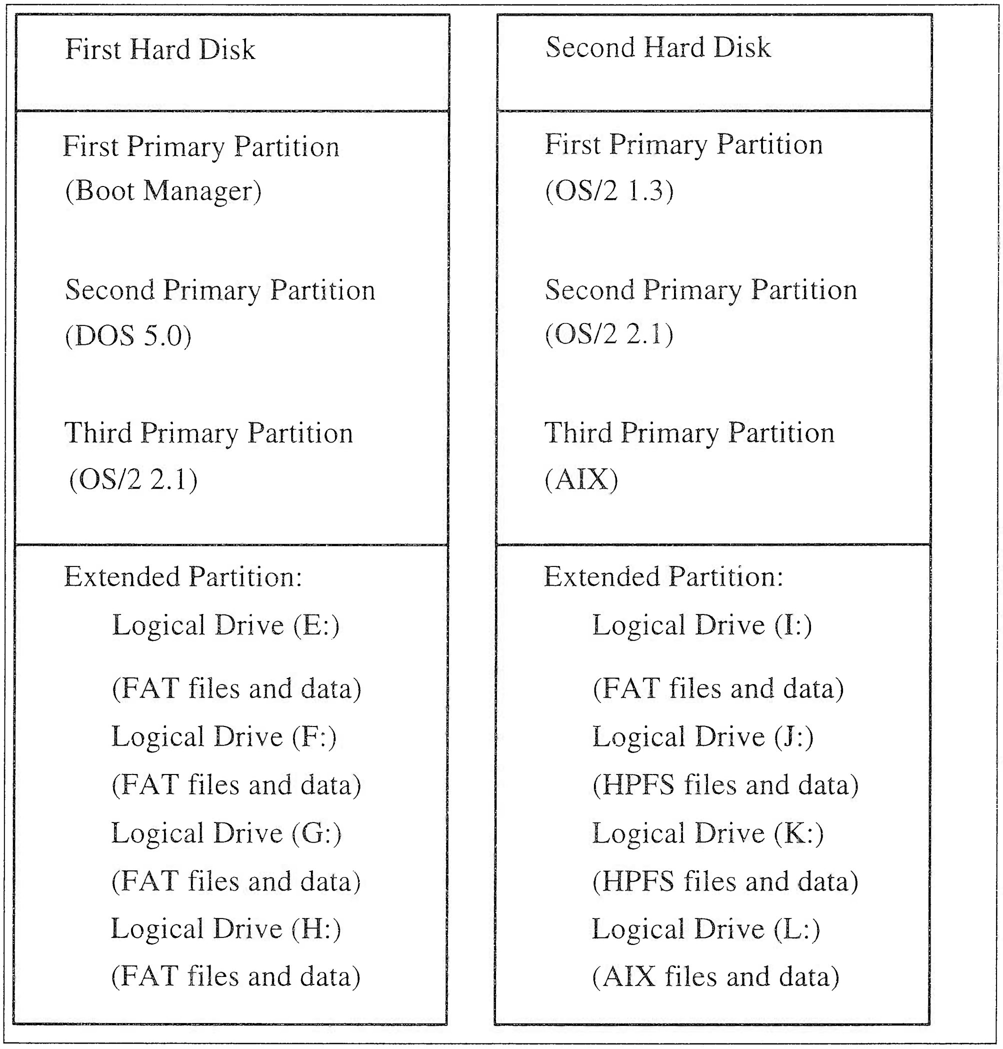


Figure 6-4. *Storage devices and drives.*

In Figure 6-4, the logical drives managed by the HPFS and by AIX are physically grouped on the hard disk after the logical drives that are managed by FAT. This arrangement is necessary because FAT stops when it does not recognize the format of a drive.

FAT can't skip over an HPFS drive to access FAT files in a later drive. However, on a machine connected to a network, you can access a remote, or redirected, drive with FAT even though your local system uses HPFS.

Local and Remote File Systems

A local file system is stored on a local device, such as the hard disk with its disk drives and virtual drives. The operating system uses a block device driver to access local file systems and to manage I/O. When you start the computer, you are automatically linked to local file systems.

A remote file system is stored on a remote device, such as a hard disk on a network server. The operating system uses a device driver for a communications or network device to access a remote file system. To have an application access a remote file system, issue a DosFSAttach API to associate the remote file system with a drive letter. The application can then treat the remote file system as if it were local by using the drive letter.

DOS and OS/2 File Objects

The DOS and OS/2 FAT file systems support the same directory structure. Therefore, an application running in DOS can access OS/2 FAT files and directories. DOS can be either the native DOS booted from a primary partition of the hard disk or a DOS session started under OS/2. An application running in OS/2 can access DOS FAT files and directories.

The HPFS directory structure is different. Native DOS booted from a primary partition does not recognize files created with the HPFS. To give a DOS application access to HPFS files, start the application in a DOS session under OS/2 2.1. However, the DOS application may still have problems with HPFS file names if they exceed the length of eight characters for the name and three characters for the extension.

File System APIs and Data Structures

Applications can use most of the same APIs to manage both the FAT and the HPFS file systems. These APIs include those listed on the next page.

- ❑ Retrieving storage information about a file or a file system
- ❑ Attaching or detaching a file system
- ❑ Retrieving device information about an attached file system
- ❑ Using an interface between an application and a file system
- ❑ Writing a cache buffer to disk

Retrieving Storage Information about a File System

The DosQueryFSInfo API retrieves information on how much storage space is available on a disk. The number of available clusters, or units of allocation (2048 bytes), is returned. The program in Figure 6-5 retrieves and prints drive information for FAT partition D. It uses the DosQueryFSInfo API, which has the following syntax:

DosQueryFSInfo

```
DosQueryFSInfo(
                ULONG   ulDriveNumber,
                ULONG   ulFSInfoLevel,
                PVOID   FSInfoBuf,
                ULONG   ulFSInfoBufSize);
```

Parameters:

- ❑ ulDriveNumber (ULONG) input—Specifies the logical drive number to return information about. (Drive A = 1, Drive B = 2, and so on to Drive Z = 26; if DriveNumber = 0, then information about the disk on the current drive is returned.)
- ❑ ulFSInfoLevel (ULONG) input—Indicates whether to return Level 1 or Level 2 file information. (Level 1 includes file system ID, number of sectors per allocation unit, number of allocation units on the disk, number of allocation units available, and number of bytes per sector; Level 2 includes volume and serial number, length of the volume label, not counting the NULL terminator, and the volume label.)

- ❑ FSInfoBuf (PVOID) output—Indicates the address of the buffer containing the returned information in one of the following formats:

Level 1		**Level 2**	
filesysid	(ULONG)	volumeserialnum	(ULONG)
sectornum	(ULONG)	volumelength	(BYTE)
unitnum	(ULONG)	volumelabel	(CHAR)
unitavail	(ULONG)		
bytesnum	(USHORT)		

- ❑ ulFSInfoBufSize (ULONG) input—Indicates the length in bytes of the buffer, FSInfoBuf.

The program in Figure 6-5 shows the DosQueryFSInfo in action.

```
/*    PROG6B.C.    Illustration of DosQueryFSInfo.                    */

#define INCL_DOSFILEMGR                /* File system values           */
#include <os2.h>
#include <stdio.h>

typedef struct _FSInfoStruct
{
   ULONG  filesysid;
   ULONG  sectornum;
   ULONG  unitnum;
   ULONG  unitavail;
   USHORT bytesnum;
} FSInfoStruct;

VOID main (USHORT argc, PCHAR argv[])
{
   ULONG   DriveNum;                  /* Drive number                 */
   ULONG   FSInfoLev;                 /* File system data             */
   ULONG   BufSize = 80;              /* Size of file sys buffer      */
   UCHAR   FileSysBuf[80];            /* Buffer for file sys info     */
   APIRET rc;                         /* Return code                  */

   DriveNum = 4;                      /* Drive D (A=1, B=2, C=3)      */

   FSInfoLev = FSIL_ALLOC;            /* File system allocation info */

   rc = DosQueryFSInfo(DriveNum,
                       FSInfoLev,
                       FileSysBuf,
                       BufSize);

   if (rc == 0) {
      printf("DosQueryFSInfo executed successfully.\n");
      printf("   File system ID is %d.\n",
         ((FSInfoStruct *) FileSysBuf)->filesysid);
      printf("   Number of sectors per allocation unit
        is %d.\n", ((FSInfoStruct *) FileSysBuf)->sectornum);
      printf("   Number of allocation units is %d.\n",
        ((FSInfoStruct *) FileSysBuf)->unitnum);
      printf("   Number of allocation units available %d.\n",
        ((FSInfoStruct *) FileSysBuf)->unitavail);
      printf("   Number of bytes per sector is %u.\n",
        ((FSInfoStruct *) FileSysBuf)->bytesnum);
   } else {
      printf("Error:  DosQueryFSInfo with rc = %ld\n",rc);
   }
   return;
}
```

Figure 6-5. *DosQueryFSInfo (PROG6B.C).*

```
DosQueryFSInfo executed successfully.
File system ID is 0.
Number of sectors per allocation unit is 4.
Number of allocation units is 51083.
Number of allocation units available 1356.
Number of bytes per sector is 512.
```

Figure 6-6. *Output from the program in Figure 6-5 (PROG6B.C)*

Attaching or Detaching a File System

The DosFSAttach API attaches or detaches a drive to or from a remote file system on a network. When a remote file system is attached to a drive, an application can use the file system as if it were local. The program in Figure 6-7 shows how to attach a drive to the driver of a remote file system. It introduces the DosFSAttach API, which has the following syntax:

DosFSAttach

```
DosFSAttach(
             PSZ    DeviceName,
             PSZ    FSDName,
             PVOID  DataBuffer,
             ULONG  ulDataBufferLen,
             ULONG  ulOpFlag);
```

Parameters:

- DeviceName (PSZ) input—Indicates a drive, a pseudocharacter device name, or a spooled device, depending on how the OpFlag parameter is set.

- FSDName (PSZ) input—Indicates the address of the remote file system driver that will be attached or detached in the call to DosFSAttach. This pointer must be set to 0 for spooled objects.

- DataBuffer (PVOID) input—Indicates the address of the file system driver or spooler structure, depending on how the OpFlag parameter is set.

- ulDataBufferLen (ULONG) input—Indicates the length in bytes of the DataBuffer parameter.

- ulOpFlag (ULONG) input—Indicates the operation requested:

Setting	**Action**
0	Attach (drive or device)
1	Detach (drive or device)
2	SpoolAttach
3	SpoolDetach

The program in Figure 6-7 shows DosFSAttach in action.

```
/*  PROG6C.C.   Illustration of DosFSAttach.                         */

#define INCL_DOSFILEMGR                 /* File system values         */
#define <os2.h>
#include <stdio.h>

VOID main (USHORT argc, PCHAR argv [])

UCHAR    DriveName[3];                  /* Drive letter               */
UCHAR    FileSysDriver[3];              /* File system driver         */
PVOID    AttachDataBuf;                 /* Data for attach argument   */
ULONG    AttachDataBufLen;              /* Length of the buffer       */
ULONG    Flag;                          /* Set to attach or detach    */
APIRET   rc;                            /* Return code                */

strcpy(DriveName, "Z:");                /* Drive to attach            */

strcpy(FileSysDriver,"\\netlan\\src");

AttachDataBuf = NULL;                   /* User does not supply data  */
AttachDataBufLen = 0;                   /* Zero length                */

Flag = 0;                               /* Attach (1=detach)          */
```

```
rc = DosFSAttach(DriveName,
                 FileSysDriver,
                 AttachDataBuf,
                 AttachDataBufLen,
                 Flag);

if (rc != 0) {
    printf("DosFSAttach rc = %ld", rc);
}
```

Figure 6-7. *DosFSAttach.*

Retrieving Device Information About an Attached File System

The DosQueryFSAttach API retrieves the name of a device and the name of the file system the device is attached to. This API works with both block and character devices, including single-file devices that act like character devices. The main reason an application might need to call DosQueryFSAttach is to ensure that a particular drive can be accessed.

The program in Figure 6-8 shows how to use DosQueryFSAttach to retrieve the file system that is managing an attachable device. Note that DosQueryFSAttach must use the file system data structure FSQBUFFER2 in which to return the information.

The DosQueryFSAttach API has the following syntax:

DosQueryFSAttach

```
DosQueryFSAttach(
                PSZ          DeviceName,
                ULONG        ulOrdinal,
                ULONG        ulFSAInfoLevel,
                PFSQBUFFER2  pDataBuffer,
                PULONG       DataBufferLen);
```

Parameters:

- ❑ DeviceName (PSZ) input—Indicates a drive or a character or pseudocharacter device name if FSAInfoLevel is set to 1; ignored if FSAInfoLevel is 2 or 3.

- ❑ ulOrdinal (ULONG) input—Indicates an index into the set of drives or the list of devices.

- ❑ ulFSAInfoLevel (ULONG) input—Indicates the level of information returned in the DataBuffer parameter:

Level	Name	Description
1	FSAIL_QUERYNAME	Retrieves information about the drive or device in the DeviceName parameter.
2	FSAIL_DEVNUMBER	Retrieves information about the item specified by the Ordinal parameter (a device); ignores the DeviceName parameter.
3	FSAIL_DRVNUMBER	Retrieves information about the item specified by the Ordinal parameter (a drive); ignores the DeviceName parameter.

- ❑ pDataBuffer (PFSQBUFFER2) output—Indicates the address of the return buffer, which contains the following fields:

Name	Data Type	Description
iType	(USHORT)	Type of item, where: 1 = resident character device. 2 = pseudocharacter device. 3 = local drive. 4 = remote drive attached to the file system driver.
cbName	(USHORT)	Length of the item's name in bytes, not counting the NULL terminator.
cbFSDName	(USHORT)	Length of file system attach information in bytes.
szName	(UCHAR)	Name of the item.
Name	**Data Type**	**Description**
szFDSName	(UCHAR)	Name of the file system driver to which the item is attached.
rgFSAData	(UCHAR)	Data about the file system attach returned by the file system driver.

- ❑ DataBufferLen (PULONG) input or output—On input, this parameter specifies the length of DataBuffer; on output, it contains the length of the data returned in DataBuffer.

Using an Interface Between an Application and a File System

The DosFSCtl API allows an application to access a file system that is not attached to a name that the operating system knows (the name is not in the file system name space). This API provides a standard interface between an application and a file system driver, which handles the I/O operations of the file system (either FAT or HPFS). When an application calls DosFSCtl, the operating system passes the control information it obtains to the dynamic link library of the file system.

DosFSCtl

```
DosFSCtl(
         PVOID    DataArea,
         ULONG    ulDataLengthMax,
         PULONG   DataLengthInOut,
         PVOID    ParmList,
         ULONG    ulParmLengthMax,
         PULONG   ParmLengthInOut,
         ULONG    ulAPICode,
         PSZ      RouteName,
         HFILE    FileHandle,
         ULONG    ulRouteMethod);
```

Parameters:

- ❑ DataArea (PVOID) input—Indicates the address of the area in which the requested information will be returned.
- ❑ ulDataLengthMax (LONG) input—Indicates the length of the DataArea parameter in bytes.
- ❑ DataLengthInOut (PULONG) input or output—Contains one of the following pointers:

On input—Length in bytes of the information passed to the file system driver in DataArea.

On output—Length in bytes of the information returned by the file system driver in DataArea.

- ❑ ParmList (PVOID) input—Indicates the address of the parameter list; parameters depend on the command issued.

- ❑ ulParmLengthMax (ULONG) input—Indicates the length in bytes of the ParmList parameter; must be larger than ParmLengthInOut on output.

- ❑ ParmLengthInOut (PULONG) input or output—Contains one of following pointers:

  On input— Length in bytes of the parameters passed to the file system driver in ParmList.

  On output—Length in bytes of the parameters returned by the file system driver in ParmList.

- ❑ ulAPICode (ULONG) input—Indicates the selection of an API associated with a remote file system driver when a bit is set in the following ranges:

Range	**Description**
0x0000 - 0x7FFF	Reserved for operating system use.
0x8000 - 0xBFFF	Processing is to be handled by the local file system driver.
0xC000 - 0xFFFF	Processing is to be exported to the server.

The APICode parameter can have one of the following values:

Value	**Name**	**Description**
1	FSCTL_ERROR_INFO	Gets error information from the file system driver.

2	FSCTL_MAX_EASIZE	Gets the maximum size of an extended attribute.

- ❑ RouteName (PSZ) input—Indicates the address of the item on which action is to be taken: either the file system driver's name or the path name of the file or directory.

- ❑ FileHandle (HFILE) input—Indicates the handle for the file or device.

- ❑ ulRouteMethod (ULONG) input—Indicates how the request will be routed, where:

Value	**Name**	**Description**
1 =	FSCTL_HANDLE	The file handle controls routing.
2 =	FSCTL_PATHNAME	The path name controls routing.
3 =	FSCTL_FSDNAME	The file system driver controls routing.

Writing a Cache Buffer to Disk

To improve performance, data written to a file is often cached in memory and not actually written to disk until later. Caching explains why the output file of a program may be empty after the program traps, even though it had issued one or more printf commands. When the program traps, the cached data is lost.

The DosResetBuffer API forces the cached buffers to disk. This procedure is also known as flushing a file. The DosResetBuffer API usually operates with one specific file handle so that only the buffers associated with that file handle are flushed. When a single file is flushed with this API, the file remains open in its directory entry. All the files associated with the calling application can be flushed by setting the file handle to 0xFFFFFFFF.

The program in Figure 6-8 opens a file named TEST.OUT, writes "All work and no play makes Jack a dull boy" 1000 times, resets the buffer (flushing any remaining text to the file), displays the file size, and closes the file. If TEXT.OUT already exists, the program won't work, since it doesn't check to see if the file exists. The program in Figure 6-8 introduces the DosResetBuffer API, which has the following syntax:

DosResetBuffer

```
        DosResetBuffer(
                       HFILE  FileHandle);
```

Parameter:

- ❑ FileHandle (HFILE) input—Indicates the file handle that points to the buffers to be written to disk.

The program in Figure 6-8 shows the DosResetBuffer API in action:

```
/*   PROG6D.C.   Illustration of DosResetBuffer.                       */

#define INCL_DOSFILEMGR                /* File system values           */
#define INCL_DOSERRORS                 /* Base error codes             */
#include <os2.h>
#include <stdio.h>

VOID main (USHORT argc, PCHAR argv[])
{
   HFILE         FileHandle;
   UCHAR         FileName[20];
   ULONG         Action;
   ULONG         ActionTaken;
   UCHAR         writeBuffer[50];
   ULONG         bytesWritten;
   ULONG         FileInfoLev;           /*  Level of info to return */
   FILESTATUS    FileInfoBuffer;        /*  Buffer for information  */
   ULONG         FileInfoBufferSize;    /*  Size of the buffer      */
   APIRET        rc;
   ULONG         i;

   strcpy(FileName,"TEST.OUT");

   rc = DosOpen(FileName,
                &FileHandle,
                &ActionTaken,
                4000,                   /* File Size                    */
                FILE_NORMAL,
                OPEN_ACTION_CREATE_IF_NEW,
                OPEN_SHARE_DENYNONE | OPEN_ACCESS_READWRITE,
                0L );                   /* No extended attributes   */

   if (rc == 0) {
      printf("DosOpen for %s executed successfully.\n",FileName);
   } else {
      printf("Error:  DosOpen for %s with rc = %ld\n",
         FileName,rc);
      if (rc == ERROR_OPEN_FAILED) {
```

```
      printf("   Check to see if the file already exists
             or is write protected.\n");
   }
   return;
}

strcpy(writeBuffer, "All work and no play makes Jack a
   dull boy.");

for (i=1;i<1000;i++) {
   rc = DosWrite(FileHandle,
                 writeBuffer,
                 sizeof(writeBuffer),
                 &bytesWritten);
   if (rc != 0) {
      printf("Error:  DosWrite for %s with rc = %ld\n",
         FileName,rc);
      break;
   }
}

rc = DosResetBuffer(FileHandle);

if (rc == 0) {
   printf("DosResetBuffer for %s executed
      successfully.\n",FileName);
} else {
   printf("Error:  DosResetBuffer for %s with rc = %ld\n",
      FileName,rc);
}

FileInfoLev = 1;                          /* Return Level 1 info     */
FileInfoBufferSize = sizeof(FILESTATUS);            /* Set size    */

rc = DosQueryFileInfo(FileHandle,
                      FileInfoLev,
                      &FileInfoBuffer,
                      FileInfoBufferSize);

if (rc == 0) {
   printf("DosQueryFileInfo for %s executed
      successfully.\n",FileName);
   printf("   File size is %d.\n",FileInfoBuffer.cbFile);
} else {
   printf("Error:  DosQueryFileInfo for %s with
      rc = %ld\n",FileName,rc);
}

rc = DosClose(FileHandle);

if (rc == 0) {
   printf("DosClose for %s executed successfully.\n",
      FileName);
} else {
      printf("Error:  DosClose for %s with rc = %ld\n",
         FileName,rc);
   }

   return;
}
```

Figure 6-8. *DosResetBuffer (PROG6D.C).*

```
DosOpen for TEST.OUT executed successfully.
DosResetBuffer for TEST.OUT executed successfully.
DosQueryFileInfo for TEST.OUT executed successfully.
File size is 49950.
DosClose for TEST.OUT executed successfully.
```

Figure 6-9. *Output from the program in Figure 6-7 if the file doesn't exist.*

Figure 6-10 shows the output that would happen if the file already exists:

```
Error:  DosOpen for TEST.OUT with rc = 110
Check to see if the file already exists or is write
protected.
```

Figure 6-10. *Output from the program in Figure 6-8 if the file already exists.*

FILE NAMES AND NAMING CONVENTIONS

When applications or their users create names for directories and files, they must observe the following rules, whether the file system is FAT or HPFS:

- Do not use characters reserved for use by the operating system. These include:

  The greater than symbol (>)
  The less than symbol (<)
  The colon (:)
  The quotation mark (")
  The slash (/)
  The back slash (\)
  The or symbol (|)

- Do not use a character in the ASCII range from 0 through 31.

- ❑ Process a path as a null-terminated string, as determined by the DosQuerySysInfo API.
- ❑ Allow the comparison of file names without regard to uppercase and lowercase distinctions.
- ❑ Use only / and \ as path separators.
- ❑ Use a period (.) to separate components in a directory or file name. Note that HPFS file names can contain more than two components.
- ❑ Use one period for the current directory and two periods (..) for the parent of the current directory as directory components in a path. You can use this both in PATH statements and in CD commands.

An application using the HPFS must observe these additional rules:

- ❑ The blank character is significant in file names. For example, PRINT .FLS refers to a different file than PRINT.FLS, which is a different file than PR INT.FLS.
- ❑ The following characters are valid in HPFS file names:

  The plus symbol (+)
  The equal sign (=)
  The semicolon (;)
  The comma (,)
  The left square bracket ([)
  The right square bracket (])

In addition, an application using the HPFS can take advantage of long file names, up to 255 characters including the null terminator. To do so, the application must include a NEWFILES declaration in its module definition file. This statement tells the linker to set a bit in the header of the executable file to recognize long file names.

If an application that does not recognize long file names attempts to access a file with a long name, certain file APIs behave in one of the following two ways:

- ❑ APIs in this list either fail or return unexpected results because these APIs cannot find a long file name. For applications running under OS/2 FAT, an error is returned.

```
DosCopy
DosCreateDir
DosDelete
DosDeleteDir
DosFindFirst
DosFindNext
DosFSAttach
DosMove
DosOpen
DosQueryFSAttach
DosQueryPathInfo
DosSearchPath
DosSetPathInfo
```

- ❑ APIs in this list pass long file names to all applications so that they can use all the directories.

```
DosQueryCurrentDir
DosSetCurrentDir
```

Names Reserved for Devices

The following names are reserved for devices and should not be used for file names:

CLOCK$	The system clock
COM1 through COM4	Serial ports 1 through 4
CON	The console keyboard and screen
KBD$	The keyboard
LPT1 through LPT3	Parallel printers 1 through 3
MOUSE$	The mouse
NUL	A dummy device that doesn't exist
POINTER$	A pointer drawing device
PRN	The default printer (LPT1)
SCREEN$	The screen

File Attributes

The attributes of a file are associated with the file from the time it is created until it is deleted. Common attributes include Read-Only, System, Archived, and Hidden. A file can be one of several types, as governed by the attributes in Table 6-2:

File Attribute	Defined Constant	Description
Normal File	FILE_NORMAL	Can accommodate typical directory APIs, such as search and list.
Read-only File	FILE_READONLY	Cannot be written to.
Hidden File	FILE_HIDDEN	Cannot be displayed in a normal directory listing.
System File	FILE_SYSTEM	Cannot be found in a normal directory search.
Archived File	FILE_ARCHIVED	Marked to show whether the file has changed since its last use.

Table 6-2. *File attributes.*

An application can set an attribute with the DosSetFileInfo API and can retrieve information about a file attribute with the DosQueryFileInfo API. The program in Figure 6-13 uses TEST.OUT from the preceding example and changes the attributes of the file. The program first opens the file in write mode and exclusively locks other files from accessing it. The program then queries the Level 1 file information and changes the date and time to which the file was last written to all ones. If these values are all zeroes, however, no change will take place. Also, if the file is on a FAT system, the last access and creation dates and times cannot be changed, only the last write and date time. The program in Figure 6-15 introduces the DosSetFileInfo API, which has the following syntax:

DosSetFileInfo

```
          DosSetFileInfo(
                         HFILE   FileHandle,
                         ULONG   uFileInfoLevel,
                         PVOID   FileInfoBuf,
                         ULONG   ulFileInfoBufSize);
```

Parameters:

- ❑ FileHandle (HFILE) input—Indicates the handle of the file on which to set file attributes.

- ❑ uFileInfoLevel (ULONG) input—Indicates the level of information to set:

Level	Name	Description
1	FIL_STANDARD	Indicates Level 1 information in the FILESTATUS3 structure.
2	FIL_QUERYEASIZE	Indicates Level 2 information in the EAOP2 (extended attributes) structure.

- ❑ FileInfoBuf (PVOID) input—Indicates the address to which the requested level of information is returned.

- ❑ uFileInfoBufSize (ULONG) input—Indicates the length of FileInfoBuf in bytes.

The program in Figure 6-11 shows the DosSetFileInfo API in action.

```
/*   PROG6E.C.   Illustration of DosSetFileInfo.                    */

#define INCL_DOSFILEMGR                  /* File system values      */
#include <os2.h>
#include <stdio.h>

VOID main (USHORT argc, PCHAR argv[])
{
   HFILE        FileHandle;              /*  File handle            */
   ULONG        Action;
   ULONG        FileInfoLevel;           /*  Level of info to return */
   ULONG        ActionTaken;
   APIRET       rc;                      /*  Return code            */
   UCHAR        FileName[20];
   FILESTATUS   FileInfoBuf;             /*   File info buffer      */
   ULONG        FileInfoBufSize;         /*   File info buffer size */
   DATETIME     DateTimeBuf;             /*   Date/Time buffer      */

   strcpy(FileName,"TEST.OUT");

   rc = DosOpen(FileName,
                &FileHandle,
                &ActionTaken,
                0,                       /*   File Size             */
                FILE_NORMAL,
                FILE_OPEN,
                OPEN_SHARE_DENYREADWRITE | OPEN_ACCESS_READWRITE,
                0L );                     /* No extended attributes */

   if (rc == 0) {
      printf("DosOpen for %s executed successfully.\n",FileName);
   } else {
      printf("Error:  DosOpen for %s with rc = %ld\n",
         FileName,rc);
      return;
   }

   FileInfoLevel = 1;                      /* Return Level 1 info   */
   FileInfoBufSize = sizeof(FILESTATUS);   /* Set size              */

   rc = DosQueryFileInfo(FileHandle, FileInfoLevel,
                      &FileInfoBuf, FileInfoBufSize);

   if (rc == 0) {
      printf("DosQueryFileInfo for %s executed
         successfully.\n",FileName);
   } else {
        printf("Error:  DosQueryFileInfo for %s with
          rc = %ld\n",FileName,rc);
   }

   FileInfoBuf.fdateLastWrite.year = 1;
   FileInfoBuf.fdateLastWrite.month = 1;
   FileInfoBuf.fdateLastWrite.day = 1;
   FileInfoBuf.ftimeLastWrite.hours = 1;
   FileInfoBuf.ftimeLastWrite.minutes = 1;
   FileInfoBuf.ftimeLastWrite.twosecs = 1;
   /*   Update the Level 1 information block associated             */
   /*   with the file.                                               */
   rc = DosSetFileInfo(FileHandle,
                    FileInfoLevel,
                    &FileInfoBuf,
                    FileInfoBufSize);

   if (rc == 0) {
      printf("DosSetFileInfo for %s executed
         successfully.\n",FileName);
   } else {
      printf("Error:  DosSetFileInfo for %s with
         rc = %ld\n",FileName,rc);
   }

   rc = DosClose(FileHandle);

   if (rc == 0) {
      printf("DosClose for %s executed successfully.\n",
         FileName);
   } else {
      printf("Error:  DosClose for %s with rc = %ld\n",
         FileName,rc);
   }

   return;
}
```

Figure 6-11. *DosSetFileInfo (PROG6E.C).*

```
DosOpen for C6_8.OUT executed successfully.
DosQueryFileInfo for C6_8.OUT executed successfully.
DosSetFileInfo for C6_8.OUT executed successfully.
DosClose for C6_8.OUT executed successfully.
```

Figure 6-12. *Output from the program in Figure 6-11.*

The directory listing of the file looks like this before the program is run:

```
TEST      OUT     49950   3-06-93   1:23p
```

After the program is run, the directory listing looks like this:

```
TEST      OUT     49950   1-01-81   1:01a
```

Note that the year value counts up from 1980, the beginning decade of IBM Personal Cmputers (and, oh yes, compatibles). Typically, however, an application sets attributes with the DosOpen API.

When the file is opened, the READ-WRITE attribute is set. After the file is closed, you can't delete or change it without either issuing an API call to set its attributes back to READ-WRITE or issuing:

```
ATTRIB -R MYFILE.OUT
```

If you issue ATTRIB MYFILE.OUT from the OS/2 command prompt, the follwing is displayed, indicating that the file is read only:

```
A    R   D:\c\MYFILE.OUT
```

In addition to READ-ONLY, you can set the share variable to the values in Table 6-3. .

Share Variable	**Description**
OPEN_ACCESS_READONLY	Open a file only for reading.
OPEN_ACCESS_WRITEONLY	Open a file only for writing.
OPEN_ACCESS_READWRITE	Open a file for both reading and writing.
OPEN_SHARE_DENYREADWRITE	Open a file for the exclusive use of the current application; deny read and write access to all other processes.

OPEN_SHARE_DENYWRITE	Open a file but deny write access to other processes.
OPEN_SHARE_DENYREAD	Open a file but deny read access to other processes.
OPEN_SHARE_DENYNONE	Open a file and grant read and write access to all processes.

Table 6-3. *File share variables.*

Theoretically, you can use access and share variables in any combination. However, if your application is going to write to a file, you should deny read and write access to other processes until your write operation is finished.

SUMMARY

This chapter introduces the files and file systems of OS/2 2.1. The FAT and IFS file systems are introduced, and a popular IFS, the HPFS, is discussed. Attaching to and detaching from a file system is presented File handles, names, and attributes are discussed, along with file naming conventions. Advice is also presented on how to partition your hard disk for various purposes. The following APIs are discussed :

DosQueryFileInfo
DosQueryFSInfo
DosFSAttach
DosQueryFSAttach
DosFSCtl
DosResetBuffer
DosSetFileInfo

CHAPTER 7

Extended Attributes for Files

"Sweet are the uses of adversity." —*Shakespeare*

INTRODUCTION

You can have a long and happy programming career and never have to bother with the adversities attendant upon working with extended attributes. But for the intrepid, extended attributes will let you pull off some nifty tricks that will fully repay the trouble of mastering this material. Since extended attributes are a fairly advanced programming topic, this chapter assumes that you don't know anything about them. If you happen to be an old hand at extended attributes, you can skip most of this chapter because the changes between earlier versions and OS/2 2.1 are minimal.

Using file system APIs, an application can create extended attributes that associate information with a file beyond the data that operating systems automatically maintain about files and directories. DOS and all versions of OS/2, for example, maintain Level 1 file information for you.

This information includes the name and size of the file as well as the date and time the file was created, most recently accessed, and most recently written to. Most of this information is displayed when you issue a DIR command on a directory.

An extended attribute of a file contains information that can be used by another application, the operating system, or the file system that manages the file. The purpose of extended attributes is one or more of the folllowing items:

- ❑ To make notes about the file, such as the name of its owner or its function.
- ❑ To describe the contents of the file, such as text or binary data.

- ❑ To explain the data record format of the file.
- ❑ To add data to the file, for example data to be processed only if a certain condition is true.
- ❑ To associate an icon with a file.
- ❑ To describe the type of the file.
- ❑ To associate data files with the applications that create or use the data.

The additional information in extended attributes can include almost anything, such as the name of the creator of the file, the version level, a history log, a summary of the contents, the icon associated with the file type, a graphic the file uses, and the code page of the file. The complete list of standard extended attributes that are provided by OS/2 2.1 is discussed later in this chapter. You can also create your own extended attributes. You can specify one extended attribute or several, provided that the total is 64KB or less.

Extended attributes have been available since OS/2 1.2. However, as long as you used only the File Allocation Table (FAT) file system, the extended attributes for all files in a directory were stored in a separate file with a unique name:

```
EA DATA.  SF
```

The spaces in the file name and its extension were included to make it harder for someone to erase the file by mistake. The EA DATA. SF file could take up a lot of disk space, and frequent accesses to it could degrade performance.

On partitions formatted for the High Performance File System (HPFS), extended attributes are still stored separate from the file itself, but each file has its own extended attributes file. HPFS manages the storage and maintenance automatically.

Figure 7-1 shows three extended attributes associated with the file APP1.EXE, a program that calculates the mean of a set of numbers contained in one of the extended attributes.

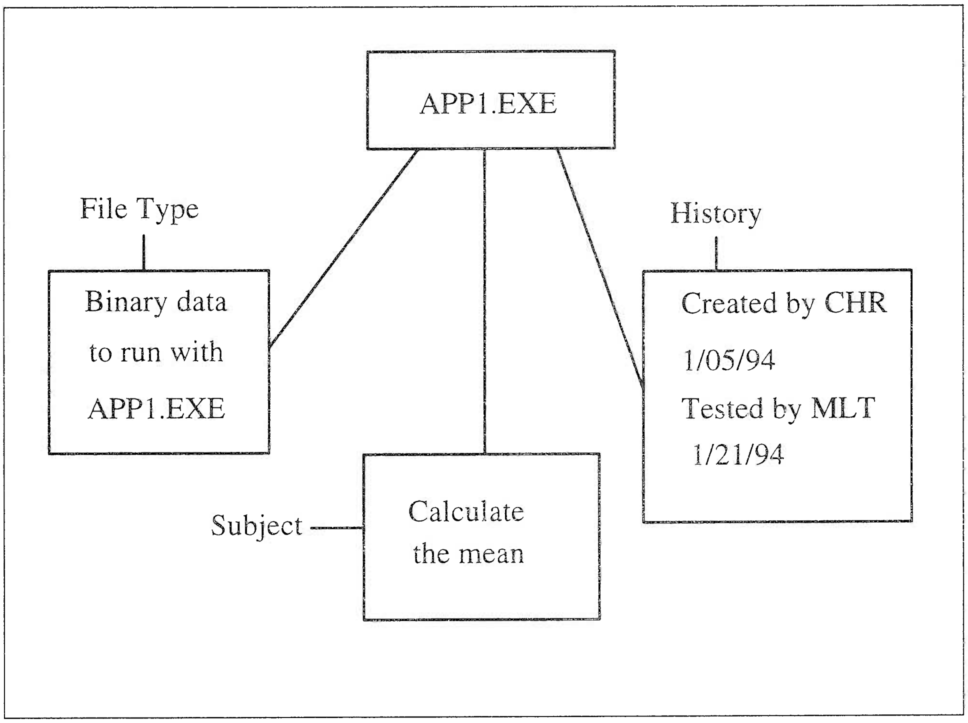

APP1.EXE
File Type
Binary data to run with APP1.EXE
History
Created by CHR 1/05/94 Tested by MLT 1/21/94
Subject
Calculate the mean

Figure 7-1. *Extended Attributes associated with an executable file.*

Standard Extended Attributes

OS/2 2.1 provides standard extended attributes with pre-defined names for you to use. These names begin with a period (.). All can contain multiple values, and some can contain multiple types.

Following are the standard extended attributes, with explanations following.

```
Most often used:

        .TYPE
        .ASSOCTABLE

Also available:

        .CODEPAGE
        .COMMENTS
        .HISTORY
        .ICON
```

```
.KEYPHRASES
.LONGNAME
.SUBJECT
.VERSION
```

Standard Extended Attribute: .TYPE

The .TYPE extended attribute is similar to the extension portion of a file name. It indicates the data type of the file it is associated with. In the HPFS, a file name and its extension can contain up to 254 characters.

The format of the .TYPE standard extended attribute is as follows, with the data type variable enclosed in curly braces:

```
EAT_ASCII { length in hex} .TYPE {name}
```

Table 7-1 shows the pre-defined file types that can go into the name variable:

```
Assembler code
BASIC code
Binary data
Bitmap
C code
COBOL code
DOS command file
Dynamic Link Library (DLL)
Executable (EXE)
FORTRAN code
Icon
Metafile (graphics)
Object code (compiled code)
OS/2 command file
Pascal code
PIF file
Plain text (ASCII)
Pointer
Printer specific
Resource file
```

Table 7-1. *Name variable contents.*

It is important for the .TYPE name to be unique because it is used when binding the data file type to the application. To make the .TYPE name unique, a naming convention is recommended. For example, you can precede the type with one or more of the following:

```
        Company_name
        Application_name
        Application_specific_name
```

The following example shows how to distinguish between the payroll for laboratory employees and those at company headquarters:

```
EAT_ASCII 0025 .TYPE ABC_Co Payroll.C Lab EXE

EAT_ASCII 0025 .TYPE ABC_Co Payroll.C HdQtrs EXE
```

The name is case-sensitive, so the uppercase and lowercase letters in the name should match the real names as they are known to the operating system. Since the .TYPE standard extended attribute is used by .ASSOCTABLE, you should get .TYPE set up first.

Standard Extended Attribute: .ASSOCTABLE

You can get an understanding of .ASSOCTABLE by viewing an Association page on the OS/2 2.1 Desktop. For example, in the Enhanced Editor, Association is one of the items you can select.

The purpose of .ASSOCTABLE is to associate data files with the applications that create or can use these data files. The .ASSOCTABLE standard extended attribute contains the file type, its extension, its icon, and an ownership flag.

If two applications both use the .ASSOCTABLE extended attribute, the calling application can retrieve this information from the called application. Another use for .ASSOCTABLE is to control, by means of the ownership flag, which application is to run when you double click on an icon for a particular data file type. The format of .ASSOCTABLE is shown in Figure 7-2, with the variables enclosed in curly braces:

```
EAT_MVMT 0000 0004 EAT_ASCII  .TYPE {name}
   EAT_ASCII   {file extension}
   EAT_BINARY  {flags}
   EAT_ICON    {icon data}
```

Figure 7-2. *Format of the .ASSOCTABLE standard extended attribute.*

The information that builds .ASSOCTABLE is contained in a table in the resource file for the application. .ASSOCTABLE can be created by the resource compiler from this table.

It has the format shown in Figure 7-3, where items in square brackets indicate optional fields:

```
ASSOCTABLE assoctable -id
BEGIN
association_name,[extension],[flags],[icon_filename]
association_name,[extension],[flags],[icon_filename]
association_name,[extension],[flags],[icon_filename]
       .
       .
END
```

Figure 7-3. *The resource file ASSOCTABLE.*

An explanation of the fields in Figure 7-3 follows:

- ❑ The association_name field is the file type contained in the .TYPE standard extended attribute. The resource compiler must be able to recognize this file type.
- ❑ The extension field indicates the file type if a file does not have a .TYPE extended attribute defined.
- ❑ The flags field can show which application will be started when the user double-clicks on the icon for a particular data file type.
- ❑ If an icon is used to represent this file type, data associated with the icon is included in the icon_filename field of FILEICON.ICO, the icon file name.

The flags field can contain the values specified in the PMWIN.H and PMWIN.INC files. For example, set the flags field to EAF_DEFAULTOWNER in the file that you want to be run when the user double-clicks on the icon for a data file type.

For each of the file types that .TYPE can contain, only one file should have its flags set to EAF_DEFAULTOWNER.

If either no file of the specified type, or else if more than one specified file, is set to EAF_DEFAULTOWNER, the operating system will not be able to determine which application to run and will display a list for the user to select from.

Another flag setting, EAF_REUSEICON, specifies to use the same icon as previously defined in the. ASSOCTABLE of the resource file.

In the following example, the resource file contains the company name, the application name, the file type, the flags, and the icon to uniquely identify the application. As Figure 7-4 shows, you can join two flags together:

```
ASSOCTABLE
BEGIN
"ABC_Co Payroll.C Dept_21 documentation", "DOC",
        EAF_DEFAULTOWNER, Payroll.ICO
"ABC_Co Payroll.C Dept_21 binary data", "BIN",
EAF_DEFAULTOWNER | EAF_REUSEICON
ABC_Co Payroll.C Dept_21 print checks", "PRT",
EAF_DEFAULTOWNER+EAF_REUSEICON
"ABC_Co Co_plan spreadsheet", "SPR", 0
"HDQRTS Corp_plan forecast", "FOR", 0
END
```

Figure 7-4. *The resource file ASSOCTABLE.*

Note that if you used a .TYPE extended attribute, you would not include the type indicators in the example above ("DOC," "BIN," "PRT," "SPR," and "FOR" in lines 1, 3, 5, 7, and 8).

Figure 7-4 shows that the payroll application can obtain certain files generated by spreadsheets at the company and the headquarters level, for example to calculate raises. However, if you clicked on these spreadsheet files, you would not start Payroll.C Dept_21 because this application is not the default owner of the spreadsheet files.

From the table in the resource file, the .ASSOCTABLE standard extended attribute would show which file types the payroll application can recognize and use, as Figure 7-5 shows:

```
EAT_MVMT    0000 0005

EAT_MVMT    0000 0004
EAT_ASCII   0025 ABC_Co Payrl.C Dept_21 documentation
EAT_ASCII   0003 DOC
```

```
EAT_BINARY flags
EAT_ASCII  icon data
EAT_MVMT   0000 0004
EAT_BINARY 0023 01110001 10101000 11000011 01010001
EAT_ASCII  0003 BIN
EAT_BINARY flags
EAT_ASCII  icon data

EAT_MVMT   0000 0004
EAT_ASCII  0025 ABC_Co Payrl.C Dept_21 print checks
EAT_ASCII  0003 PRT
EAT_BINARY flags
EAT_ASCII  icon data

EAT_MVMT   0000 0004
EAT_ASCII  001A ABC_Co Co_plan spreadsheet
EAT_ASCII  0003 SPR
EAT_BINARY flags
EAT_ASCII  icon data

EAT_MVMT   0000 0004
EAT_ASCII  0019 HDQRTS Corp_plan forecast
EAT_ASCII  0003 FOR
EAT_BINARY flags
EAT_ASCII  icon data
```

Figure 7-5. *A sample .ASSOCTABLE standard extended attribute.*

Standard Extended Attribute: .CODEPAGE

Use this extended attribute to indicate that the code page for the file differs from the code page of the application or from the system default. .CODEPAGE is usually single-value, single-type, as Figure 7-6 shows:

```
EAT_ASCII 000E .CODEPAGE 0437
```

Figure 7-6. *Example of .CODEPAGE Standard Extended Attribute.*

Standard Extended Attribute: .COMMENTS

Use this extended attribute for any notes that you want to record about the file, such as environment restrictions, memory requirements, or other important information. The .COMMENTS attribute is usually multi-value and can be of any type. Figure 7-7 provides an example.

```
EAT_MVST  0000 0003
   EAT_ASCII 001C .COMMENTS Slows down Windows
   EAT_ASCII 0015 .COMMENTS Needs 1024K
   EAT_ASCII 0019 .COMMENTS Ignore rc=10098
```

Figure 7-7. *Example of the .COMMENTS standard extended attribute.*

Standard Extended Attribute: .HISTORY

Use this extended attribute to record changes to the file. The entries must be in a standard format and must use only ASCII characters. In Figure 7-8, the variables in curly braces should be replaced with the name of the person changing the file, the action (creating, changing, or printing), and the date of the action:

```
{person} {action} {date}
```

Figure 7-8. *Format of entries in the .HISTORY standard extended attribute.*

Figure 7-9 provides an example of the .HISTORY attribute.

```
EAT_MVST 0000 0007
   EAT_ASCII 0017 Chris created 09/20/93
   EAT_ASCII 0017 LMT   printed 09/20/93
   EAT_ASCII 0017 CRF   printed 09/20/93
   EAT_ASCII 0017 DK    printed 09/21/93
   EAT_ASCII 0017 Pat   printed 09/21/93
   EAT_ASCII 0017 TRS   printed 09/22/93
   EAT_ASCII 0017 Chris changed 09/23/93
```

Figure 7-9. *Example of the .HISTORY standard extended attribute.*

To keep the .HISTORY attribute from becoming huge, your application can post a message asking the user whether to log this printing of the file.

Standard Extended Attribute: .ICON

Use this attribute to specify the icon that represents the file, such as the icon for the file when it is minimized. Since the .TYPE attribute contains an icon field, you could use it to indicate the default icon. However, the .ICON attribute is checked first. If it exists, it is used for the physical icon data. This standard extended attribute works exactly like the EAT_ICON data type listed in Table 7-2 and in the examples of the

.TYPE extended attribute. The first WORD after the name of the extended attribute contains the length in hex.

The second word contains the data, which has BITMAPARRAYFILEHEADER for its data type. This type is an array of two bitmaps, one for an icon that depends on a particular device and another that is independent of a specific device. The format of this attribute is:

```
EAT_ICON {length} {data}
```

The resource compiler creates the .ICON extended attribute by using the keyword DEFAULTICON with the file name {filename.ico} that contains the associated icon definition. Your application can use either .ASSOCTABLE or DosSetPathInfo to associate the icon with the file. Figure 7-10 contains a code sample.

Standard Extended Attribute: .KEYPHRASES

Use this extended attribute to associate text with a file that an application or database search routine will look for. Another use for .KEYPHRASES is to describe or identify the file. .KEYPHRASES is usually a multi-value, single-type attribute, as Figure 7-10 shows:

```
EAT_MVST 0000 0004 EAT_ASCII 0006 ABC_Co
   EAT_ASCII 0004 4Q93
   EAT_ASCII 001C A. R. Linton
   EAT_ASCII 0015 Washing machine sales
```

Figure 7-10. *Example of .KEYPHRASES standard extended attribute.*

If you want to set just one keyphrase, an example is:

```
EAT_ASCII 0011 .KEYPHRASE ABC_Co
```

Standard Extended Attribute: .LONGNAME

Use this extended attribute to make a short name out of a long name for the purpose of writing the file name to a file system, such as FAT, that does not support long names. The application should save the long name in the .LONGNAME attribute and should notify the user about the new short name.

When an application copies a file from FAT to HPFS, it should check to see if the .LONGNAME attribute contains a value. If so, the application should restore the long name and delete the .LONGNAME attribute. .LONGNAME is usually single-value, as in the following example:

```
    EAT_ASCII 0018 .LONGNAME special.printer.program.C
```

This name could be shortened to sprtprog.c, for example, if it were copied to a FAT disk.

Standard Extended Attribute: .SUBJECT

Use this attribute to summarize the contents or purpose of the file, up to a maximum of 40 characters. .SUBJECT is usually a single-value ASCII attribute, as the example shows:

```
    EAT_ASCII 0023 .SUBJECT Calculates Dept 21 payroll
```

Standard Extended Attribute: .VERSION

Use this attribute to record the level or version of a file, application, or DLL. The .VERSION attribute is usually single-value and can be ASCII or binary. An example is:

```
    EAT_ASCII 0016 .VERSION Payroll.C 2.1
```

THE EAOP2 DATA STRUCTURE FOR EXTENDED ATTRIBUTES

To set or query extended attributes, an application program can call an API, such as DosEnumAttribute, DosSetFileInfo, or DosQueryPathInfo. An application uses these APIs to fill in the fields in the extended attributes operation data structure (EAOP2).

This data structure is required for all operations on extended attributes. EAOP2 consists of two list structures (FEA2List and GEA2List) and an error field. The list structures contain one or more name structures (FEA2 and GEA2).

- FEA2List—A list of full extended attribute data structures. It is used to create or set extended attributes and query them; it contains the total length of the list as well as the list of FEA2 data structures. Use the FEA2List structure with certain APIs to query, delete, change, or add an extended attribute. Fields in FEA2 are required input to the DosSetFileInfo and DosSetPathInfo APIs, which create or set extended attributes. These fields are required output from the DosQueryFileInfo, DosQueryPathInfo, and DosEnumAttribute APIs, which query extended attributes.

- FEA2—The full extended attribute data structure. It comprises one element in a FEA2List structure and contains the name and the contents or value of the extended attribute, as well as the length of both fields. The name can be any characters legal for the file name. The name field cannot be of 0 length. The contents field can be set to 0, which deletes the extended attribute.

- GEA2List—A list of get extended attribute structures. It is used to return the contents of a set of extended attributes. It contains the total length of the list as well as the list of GEA2 data structures. Use the GEA2List structure with certain APIs to query an extended attribute. Fields in GEA2list are required input for APIs that query extended attributes (the APIs are DosEnumAttribute, DosQueryFileInfo, and DosQueryPathInfo).

- GEA2—The get extended attribute data structure. It comprises one element in a GEA2List structure and contains the name and the contents or value of the extended attribute, as well as the length of both fields. The name field cannot be of zero (0) length. The contents field can be set to 0, which deletes the extended attribute.

EAOP2

```
typedef struct _EAOP2 {
                            PGEA2LIST  pfpGEA2List;
                            PFEA2LIST  ppfpFEA2List;
                            ULONG      uloERROR;
                          } EAOP2;
```

Parameters:

- pfpGEA2List—Contains the list of GEA2 structures.

- ❑ pfpFEA2List—Contains the list of FEA2 structures.
- ❑ uloERROR—Contains the offset of the FEA error.

FEA2List

```
typedef struct _FEA2LIST {
                            ULONG   ulcbList;
                            FEA2    list[1];
                          } FEA2LIST;
```

Parameters:

- ❑ ulcbList—Contains the total number of bytes in the structure, including the full list.
- ❑ list[1]—Contains the variable-length FEA structures.

FEA2

```
typedef struct _FEA2 {
                      ULONG  uloNextEntryOffset;
                      BYTE   bfEA;
                      BYTE   bcbNAME;
                      USHORT uscbValue;
                      CHAR   chszName[1];
                    } FEA2;
```

Parameters:

- ❑ uloNextEntryOffset—The offset to the next entry in the array (FEA2List).
- ❑ bfEA—Contains flags.
- ❑ bcName—The length of the name field excluding the NULL terminator.
- ❑ uscbValue—Contains the length of the value or content.
- ❑ chszName—Contains the name of the extended attribute.

GEA2List

```
typedef struct _GEA2LIST {
                            ULONG    ulcbList;
                            GEA2     list[1];
                          } GEA2LIST;
```

Parameters:

- ❑ ulcbList—Contains the total number of bytes in the structure, including the full list.
- ❑ list[1]—Contains the size of a structure of variable length (GEA2).

GEA2

```
typedef struct _GEA2 {
                        ULONG   uloNextEntryOffset;
                        BYTE    bcbName;
                        CHAR    chszname[1];
                      } GEA2;
```

Parameters:

- ❑ uloNextEntryOffset—Contains the offset in bytes to the next entry.
- ❑ bcbNameontains the length of the name, excluding the NULL terminator.C
- ❑ chszname[1]—Contains the name of the extended attribute.

COMPONENTS OF AN EXTENDED ATTRIBUTE

An extended attribute consists of two components, its name and its contents (also known as its value). The name component can be any NULL-terminated string that uses the same character set as the file name. Standard extended attributes have pre-defined names that you can use, or you can create your own.

The contents or value component of an extended attribute consists of the following elements:

- ❑ The data type of the contents, contained in the first WORD of the contents component.

- ❑ The length of the contents, contained in the second WORD of the contents component except for multi-value extended attributes, which are explained in the text following Table 7-2.

- ❑ The actual contents of the extended attribute (such as ASCII text, binary data, icons, and bitmaps).

Table 7-2 describes the data-type indicator in the contents component. The columns represent the following:

- ❑ Data type, a NULL-terminated string using the same character set as file names; EAT in the name represents E(xtended) A(ttribute) T(ype).

- ❑ Hexadecimal value that represents the data type, ranging from FFDD to FFFE.

- ❑ Description of the data type.

Data Type	**Hex**	**Description**
EAT_ASN1	FFDD	ASN.1 field data. The International Standards Organization (ISO) requires this type to be associated with multi-value data streams.
EAT_ASCII	FFFD	ASCII text.
EAT_BINARY	FFFE	Binary data.
EAT_BITMAP	FFFB	Bitmap data.
EAT_EA	FFEE	ASCII name of a different extended attribute associated with this extended attribute but stored in another location; a nested extended attribute.
EAT_ICON	FFF9	Icon data.
EAT_METAFILE	FFFA	Metafile data, containing a definition of the contents of a graphic or picture.

Data Type	**Hex**	**Description**
EAT_MVMT	FFDF	M(ulti)-V(alue) M(ulti)-T(ype) data; two or more consecutive content fields with different data types.
EAT_MVST	FFDE	M(ulti)-V(alue) S(ingle)-T(ype) data; two or more consecutive content fields with the same data type.

Table 7-2. *Extended attribute data types.*

The hex values of 0x8000 and higher are reserved. However, you can use values between 0x0000 and 0x7FFF to create your own extended attributes.

The length of the contents of the extended attribute is expressed in hexadecimal, as Figure 7-11 shows:

```
EAT_ASCII 000B  HISTORY_LOG
   EAT_BINARY   0011  10010010  00110000
   EAT_ICON     0001
```

Figure 7-11. *Extended attribute lengths in hexadecimal.*

The multi-value data types let you store more than one piece of information or more than one type of information in the same extended attribute. The three data types that can contain multiple values are the following:

- ❑ MVST (multi-value single-type)
- ❑ MVMT (multi-value multi-type)
- ❑ ASN1 (multi-value data streams)

These data types provide a field to indicate the code page of the extended attribute if it is different from the code page of the file itself. For example, an application written in the Swedish code page can have a comment written in the Kanji code page for Japanese users, or vice versa.

The code page is contained in the first WORD after the name of the data type. If you leave the value at 0000, which is the default, the code page of the extended attribute is the same as the code page of the file itself.

EAT_MVST

Figure 7-12, using multi-value single-type, is an ASCII comment showing that a communications application is configured to send files to five cities.

```
EAT_MVST 0000 0005 EAT_ASCII  000C Buenos Aires
                              0006 London
                              0004 Rome
                              000D San Francisco
                              0008 Yokohama
```

Figure 7-12. *Example of a multi-value, single-type extended attribute with the same code page.*

In Figure 7-12, the first field after the data type indicator is 0000, which is the default. It shows that the extended attribute is in the same code page as the file. The next field contains 0005 to show that a list of five items follows. The next field shows that the items in the list are ASCII. The five numeric fields show the length of each name in hex. Another example, Figure 7-13, contains icons for code page 437 when the application is in a different code page.

```
EAT_MVST 0437 0002 EAT_ICON 0001
               0001
```

Figure 7-13. *Example of a multi-value, single-type extended attribute with different code pages*

EAT_MVMT

Figure 7-14, an example of multi-value multi-type, is an ASCII comment fort an icon field when the file and extended attribute use the same code page.

```
EAT_MVMT 0000 0002 EAT_ASCII 0012 Double click icon.
                   EAT_ICON  0002
```

Figure 7-14. *Example of an multi-value, multi-type extended attribute with the same code page.*

Another example, Figure 7-15, shows how to indicate that an icon changes when it is run in code page 850.

```
EAT_MVMT 0850 0004 EAT_ASCII 000C Changes from
```

```
EAT_ICON  0001
EAT_ASCII 0002 to
EAT_ICON  0001
```

Figure 7-15. *Example of a multi-value, multi-type extended attribute with different code pages.*

Nested Extended Attributes

You can include one extended attribute inside another extended attribute. EAT_EA points to the location of the nested extended attribute. In Figure 7-16, EAT_EA points to a file containing data that is to be used if a value is false:

```
EAT_MVMT 0000 0003 EAT_ASCII 000D Run if A = 0.
                    EAT_EA    0009 DATA.FILE
                   EAT_ASCII 000F Switch was run.
```

Figure 7-16. *Example of a nested extended attribute.*

DATA.FILE would have its own statement, for example:

```
EAT_BINARY  0011 00101010 10110001
```

Defining and Managing Extended Attributes

Use file system APIs to define extended attributes on a file. The following file system APIs operate on extended attributes:

- ❑ DosFindFirst and DosFindNext can search for and return specific extended attributes.
- ❑ DosOpen can create or open files with specific extended attributes.
- ❑ DosQueryFileInfo, using a file handle, and DosQueryPathInfo, using a file name, can read specific extended attributes. DosEnumAttribute can use either the handle or the file name to return information about extended attributes.
- ❑ DosSetFileInfo and DosSetPathInfo can set or change extended attributes.

An application can also define extended attributes for a directory by using the DosCreateDir API. You can protect extended attributes so that two or more processes that are operating on a file at the same time won't leave unexpected results. Two access protection schemes are available:

- ❑ Access permission based on a handle is set by the sharing mode of the file associated with the extended attributes. An application can query attributes with DosQueryFileInfo if the file is open for reading. An application can set attributes with DosSetFileInfo if the file is open for writing.

- ❑ Access permission based on a path is set by adding the file to the sharing mode that is established for the length of the access. An application can query attributes with DosQueryPathInfo if the file is open for reading and permission is set to DENY_WRITE. An application can set extended attributes with DosSetPathInfo if the file is open for writing and permission is set to DENY_READ_WRITE. An application should use DosEnumAttribute only after a file is opened in DENY_WRITE mode.

If another application that has conflicting sharing rights is accessing the extended attributes before your application opens the file, your application will return a non-zero return code.

Setting Extended Attributes

The program in Figure 7-18 sets extended attributes on an ASCII file.

```
/*******************************************************************/
/*   easet.c                                                       */
/*   a routine for setting extended attributes on any file         */
/*   given that the attribute is of type EAT_ASCII                 */
/*******************************************************************/

#define INCL_DOS
#define INCL_NOPM
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/*   structures                                                    */

typedef struct _FEAlist
{
   ULONG   oNextEntryOffset;                        /* New field */
   BYTE      fEA;                                   /* Flag byte */
   BYTE      cbName;
   USHORT    cbValue;
   CHAR     *szName;
   CHAR     *aValue;
   struct _FEAlist *next;
} FEA_list;

/*  function prototypes                                          */

HFILE openFile(char *filename);
VOID allocMem(PVOID *pvMessage, ULONG ulSize);
void SetEAOnFile(HFILE FileHandle, char *name, char *value);

/* globals                                                       */

FEA_list *FEA_list_begin;

/* defines                                                       */

#define MAX_EAS 500L

/* begin program                                                 */

int main (int argc, char **argv)
{
    HFILE FileHandle;
    char *name;
    char *value;

    if (argc < 4)  {                 /* default: current directory */
       printf("usage: %s <filename> <EA name> <EA value>
          [<EA name> <EA value>] ...\n", *argv);
       return(1);
    } else { argc--;                /* decrement for filename open */
       if(FileHandle = openFile(*++argv)) {               /* returns  */
                                 /* HFILE on success, 0 on failure */
           printf("filename: %s\n",*argv);
           while ((argc-=2) > 0) {
              name = *++argv;
              value= *++argv;
              SetEAOnFile(FileHandle, name, value);
           }
           DosClose(FileHandle);
       } else {
           printf("could not open file %s\n",*argv);
           return (1);
       }
    }
    return 0;
}

void SetEAOnFile(HFILE FileHandle, char *name, char *value)
```

```
{
    EAOP2     eaop2;
    APIRET    rc = 0;
    ULONG     ulEASize;
    USHORT     * pEAData;

  /*****************************************************************/
  /* the factor 8 in the following ulEASize assignment            */
  /* is from the following:                                        */
  /*         4 bytes for oNextEntryOffset                          */
  /*         1 byte for usFlags                                    */
  /*         1 byte for cbName                                     */
  /*         2 bytes for cbValue                                   */
  /*****************************************************************/

  /*****************************************************************/
  /* the ulEASize is big enough for the name, the value           */
  /* the 8 bytes listed above                                      */
  /* 4 bytes that go with the attribute value.                     */
  /*****************************************************************/

    ulEASize = 8 + strlen(name) + 1 + strlen(value) + 4;
    eaop2.fpFEA2List = malloc(ulEASize);
    eaop2.fpFEA2List->list->oNextEntryOffset = 0;
    eaop2.fpFEA2List->list->cbName = strlen(name);
    eaop2.fpFEA2List->list->cbValue = strlen(value)+4;


  /* set the pointer to a location one byte beyond the end          */
  /* of the 'name' field                                             */

    pEAData = (USHORT *)
                 ((PBYTE)eaop2.fpFEA2List->list->szName +
                 eaop2.fpFEA2List->list->cbName + 1);

  /* now set the data related to the attribute value: the           */
  /* type, the size, and valueData                                   */

    *pEAData = EAT_ASCII;                /* the EA type is always   */
                                         /* ASCII in this example   */
    pEAData = (USHORT *)((PBYTE)pEAData + 2);
    *pEAData = strlen(value);
    pEAData = (USHORT *)((PBYTE)pEAData + 2);
    memcpy(pEAData, value, strlen(value));

    strcpy(eaop2.fpFEA2List->list->szName, name);/*NULL termination */
    eaop2.fpFEA2List->cbList = ulEASize;

    rc = DosSetFileInfo(FileHandle, FIL_QUERYEASIZE,
            &eaop2, ulEASize);
    if(rc)
      printf("unable to set EA on this file. RC=%d\n", rc);
    else
      printf("successfully set EA \"%s\"  to ASCII value
          of: \"%s\"\n",name, value);
    return;
}

VOID allocMem (PVOID *ppv, ULONG cb)
{
    BOOL failed;

    failed =(BOOL) DosAllocMem(ppv, cb, fPERM|PAG_COMMIT);
    if (failed) {
       fprintf(stderr,"ERROR: Memory is full\n");
       *ppv = NULL;
       exit(1);
    }
    return;
}

/*********************************************************************/
/* openFile                                                          */
/* returns file handle on successful open                            */
/*     or returns 0L on failure                                      */
/*********************************************************************/

HFILE openFile(char *filename)
{
    ULONG     ulActionTaken;
    APIRET    rc;
    HFILE      hfile;

/*********************************************************************/
/* Open the file for Extended Attribute access.  The file            */
/* must be opened for read access with a deny-write                  */
/* sharing mode.                                                     */
/*********************************************************************/

    rc = DosOpen(filename,
                &hfile,
                &ulActionTaken,
                100L,
                FILE_NORMAL,  /* was READONLY */
                OPEN_ACTION_FAIL_IF_NEW
                   | OPEN_ACTION_OPEN_IF_EXISTS,
                OPEN_SHARE_DENYWRITE
                   | OPEN_ACCESS_READWRITE,
                0L);
    return(rc?0L:hfile);          /* reverse the logic of the Dos   */
                                   /* API so that a return of true  */
                                   /* indicates success             */
```

Figure 7-18. *Setting extended attributes on a file.*

Viewing Extended Attributes

The program in Figure 7-19 lets you view existing extended attributes on an ASCII file.

```
/**********************************************************************/
/* eaview.c                                                           */
/* a routine for viewing extended attributes                          */
/* Prints out the extended attribute name,                            */
/*           the size of the EA value                                 */
/*       and with certain attributes it prints                        */
/*        out the value itself                                        */
/* given that the attribute is of type EAT_ASCII                      */
/**********************************************************************/

#define INCL_DOS
#define INCL_NOPM
#include <os2.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Function prototypes.                                               */

BOOL readEA(char *path);
VOID allocMem(PVOID *pvMessage,
             ULONG ulSize);
VOID walkdir(CHAR *name);
VOID GetEAsFromFile(CHAR  *szFilename,
                  DENA2 *eaList,
                  ULONG ulEnumCount);
VOID PrintData(USHORT *pEAData);

/* Defines.                                                           */

#define MAX_GEA 500L

/* Begin program.                                                     */

int main (int argc, CHAR **argv)
{
    if (argc == 1)              /* Default: current directory.       */
      walkdir("*");
    else
      while (--argc > 0)
          walkdir(*++argv);
    return 0;
}

VOID walkdir(CHAR *name)
{
    APIRET retVal;
    FILEFINDBUF3 buf;
    ULONG srchcnt = 1;
    HDIR hdir=HDIR_SYSTEM;

    /*****************************************************************/
    /* Use DosFindFirst to find the first file in the               */
    /* directory.  Print the filename of the file and call          */
    /* readAE to print the EAs if it is found. Otherwise            */
    /* print an error message.                                      */
    /*****************************************************************/

       if (!(retVal = DosFindFirst(name,
                      &hdir,
                      FILE_NORMAL + FILE_HIDDEN
                      + FILE_SYSTEM + FILE_READONLY,
                      &buf,
                      sizeof(buf),
                      &srchcnt, 1L))) {
      printf("%15s ",buf.achName);
      readEA(buf.achName);


       /**************************************************************/
       /* Use DosFindNext to get the rest of the                     */
       /* filenames in the directory. Call readEA to                 */
       /* get the EAs for that file.                                 */
       /* an error message.                                          */
       /**************************************************************/

        while(!(retVal = DosFindNext(hdir,
                                  &buf,
                                  sizeof(buf),
                                  &srchcnt))) {
           printf("%15s ",buf.achName);
           readEA(buf.achName);
       }
    } else {
      printf("DosFindFirst failed. rc=%d\n",retVal);
    }

    return;
}

BOOL readEA(CHAR *path)
{
    CHAR *pAlloc=NULL;              /* Buffer for returned EA data.*/
    ULONG ulEntryNum = 1;           /* Current EA counter.         */
    ULONG ulEnumCnt;                /* Number of EAs to return.    */
    FEA2 *pFEA;                     /* Pointer to returned values. */


    /*****************************************************************/
    /* Allocate enough room for any attribute list for the          */
    /* returned EA buffer and initialize the pFEA to the            */
    /* start of this buffer.  The pFEA buffer is used to            */
/* traverse the returned buffer.                                  */
/*****************************************************************/

allocMem((PPVOID)&pAlloc, MAX_GEA);
pFEA = (FEA2 *) pAlloc;

/*****************************************************************/
/* Loop through all the EAs in the file.  A break is              */
/* issued when no EAs are left to be processed.                   */
/*****************************************************************/

 for(;;) {

   /*************************************************************/
   /* Use DosEnumAttribute to get a buffer (pAlloc)              */
   /* containing basic EA information about the EA.              */
   /* The actual data for the EA is not returned by              */
   /* this function.  The data is acquired by                    */
   /* using DosQueryFileInfo.                                    */
   /*************************************************************/

    ulEnumCnt = 1;
   if(DosEnumAttribute(1,
                  path,
                  ulEntryNum,
                  pAlloc,
                  MAX_GEA,
                  &ulEnumCnt,
                  1L) )   {
     printf("\n");
     break;                  /* An error occured. */
   }

   /*************************************************************/
   /* If the first EA for the file is not available              */
   /* then there are no EAs associated with this file.           */
   /*************************************************************/

     if (ulEnumCnt != 1) {
     if(ulEntryNum==1) {
         printf("    ::> none\n");
     } else {
         printf("\n");
     }
     break;
   }

   ulEntryNum++;


   /*************************************************************/
   /* Print the attribute name.  The first attribute             */
   /* offset on the line differently from the rest               */
   /* to account for the filename which also appears             */
```

```
        /* on the same output line.                                  */
        /************************************************************/

        if(pFEA->szName) {
           if(ulEntryNum==2) {
               printf("    %10s ", pFEA->szName);
           } else {
               printf("                        %10s ",
                 pFEA->szName);
           }
        }

        /************************************************************/
        /* Open the file, get the attribute value, and               */
        /* then print the EA values information.                     */
        /************************************************************/
        GetEAsFromFile(path, (DENA2 *)pAlloc, ulEnumCnt);
    }

    DosFreeMem(pAlloc);                 /* Free the buffer.          */
    return (TRUE);
}

VOID allocMem (PVOID *ppv, ULONG cb)
{
    BOOL failed;

    failed =(BOOL) DosAllocMem(ppv, cb, fPERM|PAG_COMMIT);
    if (failed) {
       fprintf(stderr,"ERROR: Memory is full\n");
       *ppv = NULL;
       exit(1);
    }
    return;
}

VOID GetEAsFromFile(CHAR  *szFileName,
                    DENA2 *eaList,
                    ULONG ulEnumCount)

{
    HFILE     FileHandle;
    ULONG     ulActionTaken;
    ULONG     ulFileAttributes = FILE_READONLY;
    ULONG     ulOpenFlag = OPEN_ACTION_FAIL_IF_NEW   |
                           OPEN_ACTION_OPEN_IF_EXISTS;
    ULONG     ulOpenMode = OPEN_SHARE_DENYREADWRITE |
                           OPEN_ACCESS_READONLY;
    ULONG     ulEACount;
    ULONG     ulEASize;
    EAOP2     eaop2;
    USHORT    *pEAData;
    APIRET    rc = 0;
```

```
/*************************************************************/
/* Open the file for Extended Attribute access.  The        */
/* file must be opened for read access with a               */
/* deny-write sharing mode.                                 */
/*************************************************************/

 rc = DosOpen(szFileName,
          &FileHandle,
          &ulActionTaken,
          100L,
          ulFileAttributes,
          ulOpenFlag,
          ulOpenMode,
          0L);

if (rc == 0) {
   for (ulEACount=0;ulEACount<ulEnumCount;ulEACount++) {

        /****************************************************/
        /* Allocate storage for the EAs and setup the        */
        /* GEA2LIST based on the list retrieved from         */
        /* DosEnumAttribute. The storage size is             */
        /* based on the following formula :                  */
        /*                                                   */
        /*   4 Bytes - for oNextEntryOffset                  */
        /*   1 Bytes - for usFlags                           */
        /*   1 Bytes - for cbName                            */
        /*   2 Bytes - for cbValue                           */
        /*  ---------------------------------                */
        /*   8 Bytes - Total                                 */
        /*                                                   */
        /* This value (8) is added to the length of          */
        /* the attribute name (cbName) plus 1 byte           */
        /* for the NULL terminator plus the length           */
        /* of the actual attribute value                     */
        /* information (cbValue).                            */
        /*                                                   */
        /* ------------------------------------------------- */
        /*                                                   */
        /*   8 + cbName + 1 + cbValue                        */
        /*                                                   */
        /****************************************************/

        if (rc == 0) {
          ulEASize = 8 +
                   (eaList[0].cbName + 1) +
                   eaList[0].cbValue;
          eaop2.fpGEA2List = malloc(ulEASize);
          eaop2.fpFEA2List = malloc(ulEASize);

          if ((eaop2.fpGEA2List != NULL) &&
              (eaop2.fpFEA2List != NULL)) {
              eaop2.fpGEA2List->list->
                 oNextEntryOffset = 0;
```

```
       eaop2.fpGEA2List->list->cbName =
          eaList[0].cbName;
       strcpy(eaop2.fpGEA2List->list->szName,
            eaList[0].szName);
       eaop2.fpGEA2List->cbList = ulEASize;
   } else {
       rc = -1;
   }
}

if (rc == 0) {
/*********************************************************/
/* Use DosQueryFileInfo to retrieve the Full              */
/* EA data for the EA specified in fpGEA2List.            */
/*********************************************************/

   eaop2.fpFEA2List->cbList = ulEASize;
   rc = DosQueryFileInfo(FileHandle,
                         FIL_QUERYEASFROMLIST,
                         &eaop2,
                         ulEASize * 2);

   *******************************************************/
   /* If attribute information was found then           */
   /* allocate enough storage to hold the               */
   /* data and copy the data into the new               */
   /* storage.  Pass the EA data to the                 */
   /* printing routine.                                 */
   /*****************************************************/

   if (rc == 0) {
      pEAData = malloc(eaop2.fpFEA2List->
         list->cbValue);
      if (pEAData != NULL) {
         memcpy((UCHAR *)pEAData,
                (PBYTE)eaop2.fpFEA2List +
                sizeof(FEA2LIST) +
                eaop2.fpFEA2List->list->
                  cbName,
                eaop2.fpFEA2List->list->
                  cbValue);
         PrintData(pEAData);
         free(pEAData);
      } else {
         printf("- Memory allocation error\n");
      }
   } else {
      printf("- DosQueryFileInfo RC = %d\n",rc);
   }
   free(eaop2.fpGEA2List);
   free(eaop2.fpFEA2List);
   eaList = (DENA2 *)((PBYTE)eaList +
     eaList[0].oNextEntryOffset);
} else {
```

```
              printf("- Memory allocation error.\n");
            }
        }

        DosClose(FileHandle);
      } else {
        printf("- Unable to open file - DosOpen
           RC = %d.\n",rc);
      }
      return;
}


VOID PrintData(USHORT *pEAData)
{
    CHAR      szPrintBuffer[1024];
    USHORT    usEAType;
    USHORT    usEALen;
    BYTE      bByteValue;
    ULONG     i;

    /***************************************************************/
    /* pEAData points to a section of memory that                   */
    /* contains three pieces of information for the EA.             */
    /* These are EA Type, EA length and EA data for                 */
    /* most EAs.  For multiple value EAs, this info is              */
    /* a little different.                                          */
    /*                                                              */
    /* usEAType is stored at the beginning and has a                */
    /* length of 2 bytes.  usEALen  is follows usEAType             */
    /* and also has 2 bytes.   The EA data  follows                 */
    /* the usEALen and a length of usEAlen bytes.                   */
    /*                                                              */
    /* After storing type and length we move up the                 */
    /* pEAData pointer to point to the value itself.                */
    /* In order to move up the pEAData pointer a byte               */
    /* at a time we use the (PBYTE) cast.                           */
    /***************************************************************/

    usEAType = (USHORT)*pEAData;
    pEAData = (USHORT *)((PBYTE)pEAData + 2);
    usEALen  = (USHORT)*pEAData;
    pEAData = (USHORT *)((PBYTE)pEAData + 2);

    printf("%4d - ", usEALen);

    /***************************************************************/
    /* Print EA information based on the EA type.                   */
    /***************************************************************/

  switch(usEAType) {
       case EAT_EA      :
       case EAT_ASCII          :
           strncpy(szPrintBuffer,(UCHAR *)pEAData,usEALen);
```

```
            szPrintBuffer[usEALen] = '\0';
            printf("\"%s\"\n", szPrintBuffer);
            break;

        case EAT_BINARY    :
            for (i=0;i<usEALen;i++) {
               bByteValue = (BYTE)*pEAData;
               if (bByteValue<=0x0F) {
                   printf("0");
               }
               printf("%x",bByteValue);
               pEAData = (USHORT *)((PBYTE)pEAData + 1);
            }
            printf("\"\n");
            break;


        case EAT_METAFILE  :
            printf("METAFILE Type\n");
            break;

        case EAT_BITMAP    :
        case EAT_ICON    :
            printf("ICON or BITMAP Type\n");
            break;

        case EAT_MVMT   :
        case EAT_MVST   :
            printf("MVMT or MVST multivalue Type\n");
            break;

        default : {
            printf("Unknown listType EA Type: %d\n",
               usEAType);
        }
    }
    return;
}
```

Figure 7-19. Viewing extended attributes.

Critical Extended Attributes

By default, extended attributes are not considered critical to the operation of an application. If you lost the history, comments, and minimized icons for a file, for example, the application could still run correctly. However, you can specify that an extended attribute is critical to an application. In a communications application, for example, the list of cities that files are sent to is critical because without the list, no files will be sent.

To specify that an extended attribute is critical, set bit 7 in the bfEA value of the FEA2 data structure to 1. This action prevents applications that don't recognize extended attributes at all from deleting the extended attributes. However, this action doesn't prevent deleting the file itself.

Copying or Moving Files that Have Extended Attributes

If you copy or move a file with extended attributes to an operating system or file system that doesn't support extended attributes, you can lose the extended attributes. For example, copying or moving an OS/2 2.1 HPFS file to a DOS disk, a FAT disk, or an OS/2 version earlier than 1.2 will destroy all extended attributes except for its file name, which can be stored in the .LONGNAME attribute.

SUMMARY

The advanced topic of extended attributes can be optional for many programmers. However, learning to use them gives you a way to tailor your files and applications by associating additional information with them. The file system APIs that operate on extended attributes are the following:

Name	**Description**
DosCreateDir	Creates a directory and sets extended attributes.
DosEnumAttributes	Retrieves the names and lengths of extended attributes on a file.
DosFindFirst	Finds the first file object or extended attribute name that matches the specification.
DosFindNext	Finds the next file object or extended attribute name that matches the specification.
DosOpen	Opens a file that can have extended attributes.
DosQueryFileInfo	Gets file information, including extended attributes.
DosQueryPathInfo	Gets file information for a file or subdirectory, including extended attribute information.

Name	**Description**
DosSetFileInfo	Sets or changes file information, including extended attribute information.
DosSetPathInfo	Sets or changes path information for a file or directory, including extended attribute information.

If you're familiar with Workplace Shell, you can also manage extended attributes with such APIs as WinLoadFileIcon, WinSetFileIcon, wpQueryType, and wpSetType.

Extended Attribute Data Structures

Name	**Description and Structure Declaration**
EAOP2	The E(xtended) A(ttribute) OP(eration) data structure is used for all manipulation of extended attributes.
FEA2List	The list of FEA2 data structures is used in the EAOP2 data structure for APIs that create, query, or set extended attributes.
FEA2	The F(ull) E(xtended) A(ttribute) data structure contains the name and value of an extended attribute.
GEA2List	The list of GEA2 data structures is used in the GEA2 data structure for functions that query extended attributes.
GEA2	The G(et) E(xtended) A(ttribute) data structure contains the name of an extended attribute. GEA2 is used in a GEA2List to query extended attributes.

Table 7-3. *Extended attribute data structures.*

CHAPTER 8

Memory Management

"With our eyes fixed on the future, but recognizing the realities of today...we will achieve our destiny to be as a shining city on a hill for all mankind to see."

—President Ronald Reagan

INTRODUCTION

Perhaps the biggest break between 32-bit OS/2 2.1 and the 16-bit DOS and OS/2 1.x world lies with its implementation of the flat memory model. The 16-bit world of programming uses a segmented memory model, wherein memory objects are restricted to a maximum size of 64KB. If a memory object exceeded 64KB, the application would have to manage the large object by spanning its data over multiple smaller objects and juggling the objects accordingly to access their information. The OS/2 2.1 flat memory model lifts the burden from the program (and the programmer) by treating memory as a flat or linear range of addresses, up to 4GB. If an object needs to be 1MB in size, you can either allocate that amount in one API call or allow it to grow in increments. A word processing document that expands as the user composes it would fall in the latter category.

With the virtual addressing provided by OS/2 2.1, applications can also deal with memory objects that exceed the size of physical memory on the workstation. Need 20MB in objects? No problem. Memory that is not needed, whether existing as reserved structures in a program or as executable (EXE) or dynamic linking (DLL) code, is swapped from physical memory to disk until it is needed. This swap file also stores memory objects that exceed physical memory.

The flat memory model makes the programmer's job that much easier by eliminating the need for sophisticated memory management. OS/2 2.1 takes full advantage of the memory management and paging schemes built into the Intel 80386, 80486, and Pentium processors.

However, it also allows applications developed for this model to be ported more easily to other platforms, like AIX. The memory management scheme of the programs is not tied to the memory management constraints imposed by a specific processor, such as with Intel 80286- or 8086-based machines.

OS/2 2.1 also provides a high degree of memory protection. The memory address space of processes is protected from other processes, and all the threads in a process share the memory of their parent process. An application in one process cannot access or accidentally corrupt the private memory in another.

MEMORY OBJECTS

In OS/2 1.x, memory objects could range in size from 1 byte to 64KB. They were accessed using a 16:16 scheme, which meant that a memory address was identified by a 16-bit selector and a 16-bit offset. Thus, 64KB was the maximum size that an object could be (2**16 = 65,536 = 64KB). The maximum address space that a 16-bit program could reference was 512MB.

OS/2 2.1 uses a 0:32 addressing scheme, which discards the selector and gives the offset 32-bits to use when referencing an object. Thus, a memory object can be up to 4GB (2**32), which is also the maximum range that a 32-bit address can reference.

Although the system can use the entire 4GB, applications are limited by OS/2 2.1 to 512MB. This region must accommodate the program's executable file, private and shared memory, and private and shared DLLs. Keeping the range of available process address space at 512MB allows OS/2 1.x applications to remain compatible with the OS/2 2.1 environment. The 512MB boundary is only temporary, and this restriction may be lifted in a future version of OS/2. Still, 512MB provides a plethora of addressing space for applications.

The 32-bit linear address space is shown in Figure 8-1 on the next page.

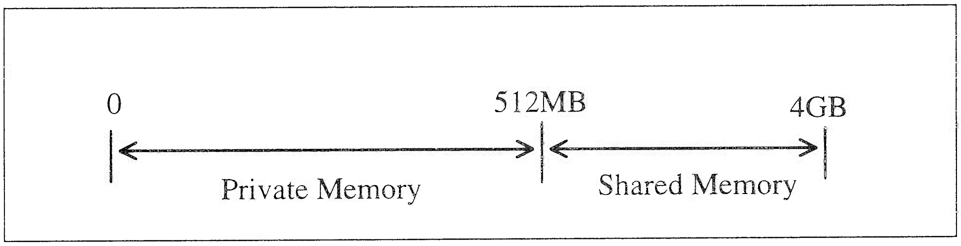


***Figure 8-1.** OS/2 2.1 32-bit addressing space.*

Whereas the process region spans the 0 to 512MB range, OS/2 2.1 uses memory from the 512MB to 4GB range as system memory. Only tasks running at the operating system privilege level can access it. Separating this region from the private memory of a user's process protects the system memory from errant addressing.

In OS/2 1.x, memory objects could be of any size (so long as it was less than 64KB). However, in OS/2 2.1 objects must be allocated in 4KB increments. These 4KB blocks are known as pages and they greatly simplify as well as increase the efficiency of memory management. (The switch from 16:16 to 0:32 addressing also eliminated the overhead of costly pointer arithmetic.)

When an object needs to be manipulated by the system, OS/2 2.1 can deal with pages of uniform length, rather than with objects of varying size. This does not mean that an application must allocate 4KB, 8KB, or some N*4KB size for an object; it can allocate a structure as small as 1 byte. However, OS/2 will reserve an entire page (4KB) to store that 1-byte object, and if (for another example) the application allocates a 10KB object, OS/2 will reserve 3 pages (12KB) for it.

If you are concerned with memory requirements, remember how pages are used for your application's memory. You could group many small memory objects into larger ones through the use of pool memory. This lessens the possibility of fragmented and unused memory. An example later in this chapter shows the use of pool memory.

For the most part, an application does not need to know whether an object needs one or N pages of storage, but when an object exceeds its allocated range, the application will not encounter a trap condition unless it also exceeds the last page boundary for that object. This is a change from OS/2 1.x where the system would immediately trap an application that attempted to go beyond the maximum range of an object. Exceeding a defined range, though not causing a trap, will corrupt data if the referenced memory object is not the sole occupant of the page.

A process can dynamically allocate and deallocate swappable linear memory from its virtual address space by using the DosAllocMem and DosFreeMem APIs.

DosAllocMem API

```
DosAllocMem(
            PPVOID BaseAddress,
            ULONG ObjectSize,
            ULONG AllocationFlags);
```

Parameters:

- ❑ BaseAddress (PPVOID) output—Essentially indicates the name of the variable to which you want the allocated memory assigned.
- ❑ ObjectSize (ULONG) input—Refers to the initial size of the object to allocate. (As mentioned earlier, the number of bytes allocated will be rounded up to the next 4K page.)
- ❑ AllocationFlags (ULONG) input—Specifies the allocation attributes and access protection for the memory object as follows:

Flag	Value	Description
PAG_READ	(0x00000001)	Grant read access.
PAG_WRITE	(0x00000002)	Grant write access (also implies read and execute access).
PAG_EXECUTE	(0x00000004)	Grant execute access (which is equivalent to read access on an Intel 80386 processor).
PAG_GUARD	(0x00000008)	Invoke the guard page fault exception condition when the page is accessed.
PAG_COMMIT	(0x00000010)	Commit all pages for the memory object. An application cannot read or write to a memory object without committed pages.

Flag	Value	Description
OBJ_TILE	(0x00000040)	Allow for 16-bit compatibility by mapping a 32-bit object to a 64KB boundary. (Only needed to insure16-bit compatibility when OS/2 raises the 512MB process virtual memory limit. OS/2 2.1 currently ignores this flag if specified.)

At a minimum the PAGE_READ, the PAGE_WRITE, or the PAGE_EXECUTE flag must be set.

The PAG_GUARD flag allows programs to register an exception handler to detect when a guard page has been accessed. The guard page can be, for example, the last committed page in a range of committed pages for an object. When the guard page fault occurs, the program's exception handler could then catch it, allocate additional memory to the object, and finally set a new guard page. This feature allows a word processor to dynamically increase the storage needed for a document as the length of its text increased. The word processor would not be forced to check each time if added text would exceed its memory range.

The DosFreeMem API is just as simple. Only the base address of the object needs to be supplied:

DosFreeMem

```
DosFreeMem (
            PVOID BaseAddress);
```

Parameters:

- ❑ BaseAddress (PVOID) input—The name of the variable object that you allocated with the DosAllocMem API.

The example program, PROG8A.C in Figure 8-2, puts the DosAllocMem and DosFreeMem APIs into action. The allocated page for the object is set to read and write access (PAG_WRITE), and the system will commit the page upon allocation (PAG_COMMIT). An application must *commit* the pages allocated for a variable (or memory object) to physical memory before data is assigned to it. Otherwise, the application will terminate with a general protection fault. Commitment need not take place at allocation time. It can be done before the object is accessed by using the DosSetMem API. (A program must make a commitment if it wants a lasting and fruitful relationship with a dynamically allocated memory object.)

```
/* prog8a.c - Show how page memory is allocated                        */

#define INCL_DOSMEMMGR              /* Include Memory Management Calls */
#define INCL_NOPMAPI                /* Don't include PM APIs           */
#include <os2.h>
#include <stdio.h>
#include <bsememf.h>                /* Flags for memory management     */

VOID main (USHORT argc, PCHAR argv[])
{
   PVOID    ObjAlloc;               /* Pointer to memory object        */
   ULONG    ObjSize;                /* Size of memory object           */
   ULONG    PageSize;               /* Size of a page                  */
   ULONG    AllocFlags;             /* Attributes of the memory object */
   APIRET   rc;                     /* Return code                     */
   int i;                           /* Counter                         */

   ObjSize = 500;                /* Will be rounded up to 4KB (1 page) */
   PageSize = 4096;

   /* Read/write access and precommit                                  */
   AllocFlags = PAG_WRITE | PAG_COMMIT;
   rc = DosAllocMem(&ObjAlloc, ObjSize, AllocFlags);

   if (rc == 0) {
      printf("DosAllocMem for ObjAlloc executed successfully.\n");
   } else {
      printf("Error:  DosAllocMem for ObjAlloc with rc = %ld\n",rc);
      return;
   }
   /* Fill and exceed the size of the memory object                    */
   strcpy(ObjAlloc,"A");
   for (i=1;i<(PageSize - 1);i++) {
      strcat(ObjAlloc,"A");
   }
   printf("ObjAlloc = %s\n",ObjAlloc);

   rc = DosFreeMem(ObjAlloc);

   if (rc == 0) {
      printf("DosFreeMem for ObjAlloc executed successfully.\n");
   } else {
      printf("Error:  DosFreeMem for ObjAlloc with rc = %ld\n",rc);
   }
   return;
}
```

Figure 8-2. *DosAllocMem and DosFreeMem (PROG8A.C).*

Figure 8-2 also demonstrates that an object is given the number of full pages (4K blocks) required for that allocated object to fit. In this example, since the ObjAllocated variable was allocated with only 500 bytes, one page will be reserved for the object. After allocation, the program begins to assign data to the variable, up to 4096 characters. (A committed page is initialized to all zeros when first accessed.)

In OS/2 1.x, once the for loop exceeded the 500th character, the program would trap, since OS/2 1.x dealt with variable sized memory objects instead of pages. As mentioned earlier, OS/2 2.1 will not trap the program until it exceeds the boundary of the last committed page for that memory object. Of course, you will not want to rely on any extra memory buffer that the remaining part of a page may provide. You will prevent errors in the future if you make sure your program doesn't access an index outside of the memory object.

Before any program terminates, it should free the memory that it had previously allocated to allow the system to reclaim it for other processes. The system actually frees an object completely when its usage count falls to zero. Shared memory objects, which will be discussed later, can have usage counts above one. This example used the DosFreeMem API to free memory that was allocated.

```
DosAllocMem for ObjAlloc executed successfully.
ObjAlloc = AAAAAAAA.........AAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
DosFreeMem for ObjAlloc executed successfully.
```

Figure 8-3. *Output from PROG8A.C in Figure 8-2.*

The next example also allocates 500 bytes of memory for the ObjAllocated variable, but instead of committing the memory at allocation time, it defers it until afterwards, using the DosSetMem API.

DosSetMem

```
    DosSetMem(
              PVOID BaseAddress,
              ULONG RegionSize,
              ULONG AllocationFlags);
```

Parameters:

- ❑ BaseAddress (PVOID) input—Specifies the name of the variable object that you allocated with the DosAllocMem API. Note that the address does not necessarily have to be the starting address. If you specify another address, within the same object, you need to vary the region size. This size will need to be smaller than the object size, and it must not exceed the edge of the object boundary.

- ❑ RegionSize (ULONG) input—Indicates the scope of the memory object to which the AllocationFlags will pertain. This value can be different from the actual object size, if the application needs to change the attributes of a smaller range of pages. The actual size is rounded up to the nearest page.

- ❑ AllocationFlags (ULONG) input—Specifies the allocation attributes for the memory object. In addition to the flags that could have been specified with the DosAllocMem API, DosSetMem also allows:

Flag	**Value**	**Description**
PAG_DECOMMIT	(0x00000020)	Decommits the specified range of pages.
PAG_DEFAULT	(0x00000400)	Uses the same access protection flags that were set when the memory was allocated.

With DosSetMem, a memory object's attributes can be changed after allocation. At a minimum, either the PAG_DECOMMIT or PAG_READ flag must be set, or one or more of the following: the PAG_WRITE, PAG_EXECUTE, or PAG_DEFAULT flag. DosSetMem does not use the OBJ_TILE flag.

The example program in Figure 8-4, PROG8B.C, first allocates, but does not commit, writeable pages for the ObjAlloc variable. The next API call to DosSetMem uses the PAG_WRITE flag that was already specified and commits the pages to physical memory. After data is assigned to the memory object, the attributes of the memory object are queried using the DosQueryMem API:

DosQueryMem

```
DosQueryMem(
            PVOID BaseAddress,
            ULONG RegionSize,
            ULONG AllocationFlags);
```

Parameters:

- ❑ BaseAddress (PVOID) input—Specifies the name of the variable object that you allocated with the DosAllocMem API.
- ❑ RegionSize (ULONG) input and output—Contains the range of bytes of the memory object that should be queried. It returns, as output, the queried size of the actual pages used by the object.
- ❑ AllocationFlags (ULONG) output—Returns an unsigned integer that can be tested against the following flags to determine the attributes of a queried region:

Flag	Value	Description
PAG_READ	(0x00000001)	Read access allowed.
PAG_WRITE	(0x00000002)	Write access allowed (also implies read and execute access).
PAG_EXECUTE	(0x00000004)	Execute access allowed (also implies read access).
PAG_GUARD	(0x00000008)	Guard page attribute is set.
PAG_COMMIT	(0x00000010)	Pages are committed.
PAG_FREE	(0x00004000)	Pages are free.
PAG_SHARED	(0x00002000)	Pages are part of a shared memory object, as opposed to a private one.
PAG_BASE	(0x00010000)	The first page in the queried region is the first page of the memory object.

Figure 8-4 shows the DosSetMem and DosQueryMem APIs in action.

```
/* prog8b.c - Allocate, but commit memory later.                 */
/* Query the allocated object too.                               */

#define INCL_DOSMEMMGR              /* Include Memory Management Calls */
#define INCL_NOPMAPI                /* Don't include PM APIs          */
#include <os2.h>
```

```
#include <stdio.h>
#include <bsememf.h>              /* Flags for memory management     */

VOID main (USHORT argc, PCHAR argv[])
{
   PVOID  ObjAlloc;          /* Pointer to memory object              */
   ULONG  ObjSize;           /* Size of memory object                 */
   ULONG  AllocFlags;        /* Attributes of the memory object       */
   ULONG  MemSize;           /* Size of memory region                 */
   ULONG  MemFlags;          /* Attribute flags of the memory region */
   APIRET rc;                /* Return code                           */

   ObjSize = 500;            /* Will be rounded up to 4KB             */
   AllocFlags = PAG_WRITE;   /* page is read/writeable                */

   rc = DosAllocMem(&ObjAlloc, ObjSize, AllocFlags);

   if (rc == 0) {
      printf("DosAllocMem for ObjAlloc executed successfully.\n");
   } else {
      printf("Error:  DosAllocMem for ObjAlloc with rc = %ld\n",rc);
      return;
   }

   /* Query the page attributes in which the object is located.     */
   MemSize = ObjSize;
   rc = DosQueryMem(ObjAlloc, &MemSize, &MemFlags);
   if (rc == 0) {
      printf("   DosQueryMem for ObjAlloc executed successfully.\n");
      printf("   MemSize = %ld\n", MemSize);
      if (MemFlags & PAG_COMMIT) {
         printf("   Page Memory has been committed.\n");
      } else {
         printf("   Page Memory has not been committed.\n");
      }                                                 /* endif */
   } else {
      printf("Error:  DosQueryMem for ObjAlloc with rc = %ld\n",rc);

   /* It's good practice to free the memory before you leave,       */
   /* even though your process' memory will be freed by OS/2 when   */
   /* the process terminates.                                       */

      rc = DosFreeMem(ObjAlloc);
      return;
   }

   /* Commit the pages for the memory object                        */

   AllocFlags = PAG_DEFAULT | PAG_COMMIT ;
   rc = DosSetMem(ObjAlloc, ObjSize, AllocFlags );

   if (rc == 0) {
      printf("DosSetMem for ObjAlloc executed successfully.\n");
   } else {
      printf("Error:  DosSetMem for ObjAlloc with rc = %ld\n",rc);
      rc = DosFreeMem(ObjAlloc);
      return;
   }

   strcpy(ObjAlloc,
    "\"There is a simple explanation for why this has happened.\"");
   printf("ObjAlloc = %s\n",ObjAlloc);

   /* Again, query the page attributes in which the object is       */
   /* located.                                                      */

   MemSize = ObjSize;
   rc = DosQueryMem(ObjAlloc, &MemSize, &MemFlags);
   if (rc == 0) {
      printf("   DosQueryMem for ObjAlloc executed successfully.\n");
      printf("   MemSize = %ld\n", MemSize);
      if (MemFlags & PAG_COMMIT) {
         printf("   Page Memory has been committed.\n");
      } else {
         printf("   Page Memory has not been committed.\n");
      }                                                    /* endif */
   } else {
      printf("Error:  DosQueryMem for ObjAlloc with rc = %ld\n",rc);
   }

   rc = DosFreeMem(ObjAlloc);

   if (rc == 0) {
      printf("DosFreeMem for ObjAlloc executed successfully.\n");
   } else {
      printf("Error:  DosFreeMem for ObjAlloc with rc = %ld\n",rc);
   }

   return;
}
```

Figure 8-4. *DosSetMem and DosQueryMem (PROG8B.C).*

This example tested to see if the region was committed using the DosQueryMem API. Other tests could have been performed as well. Lastly, the allocated memory is freed and the program terminated.

The output is shown in Figure 8-5.

```
DosAllocMem for ObjAlloc executed successfully.
   DosQueryMem for ObjAlloc executed successfully.
   MemSize = 500
   Page Memory has not been committed.
DosSetMem for ObjAlloc executed successfully.
```

```
ObjAlloc = "There is a simple explanation for why this has
   happened."
   DosQueryMem for ObjAlloc executed successfully.
   MemSize = 500
   Page Memory has been committed.
DosFreeMem for ObjAlloc executed successfully.
```

Figure 8-5. *Output from PROG8B.C in Figure 8-4.*

SHARED MEMORY

OS/2 2.1 allows memory objects to be either shared or private. Whether the former or latter, space for both will be carved out of the address space that belongs to the process which owns or accesses the memory object. As mentioned earlier, a process's virtual address space spans the 0 to 512MB range out of the 4GB of address space that is available with the 32-bit linear addressing scheme.

The *private memory* of a process, also called its private arena, is allocated from the bottom of its address space, growing upward. Its *shared memory*, or shared arena, is allocated from the top of its address space, flowing downward.

Shared memory has the same addressing range for every process. This means that the entire shared object spans the same range in every process. This allows the process to access the shared memory object without needing any special address conversion routines.

The shared memory arena includes shared memory objects and shared DLLs, although the instance data used by the shared DLLs will reside in the private arena of the process. The private memory arena also includes private memory objects and unique instances of EXEs.

Executable programs may also be shared across processes, but their unique instances will always occupy the lowest part of the private region of a process. This happens because the EXE runs first and is the essence of the process. Both arenas start with a minimum allocation of 64KB each. Figure 8-6 shows example private and shared arenas for two different processes.

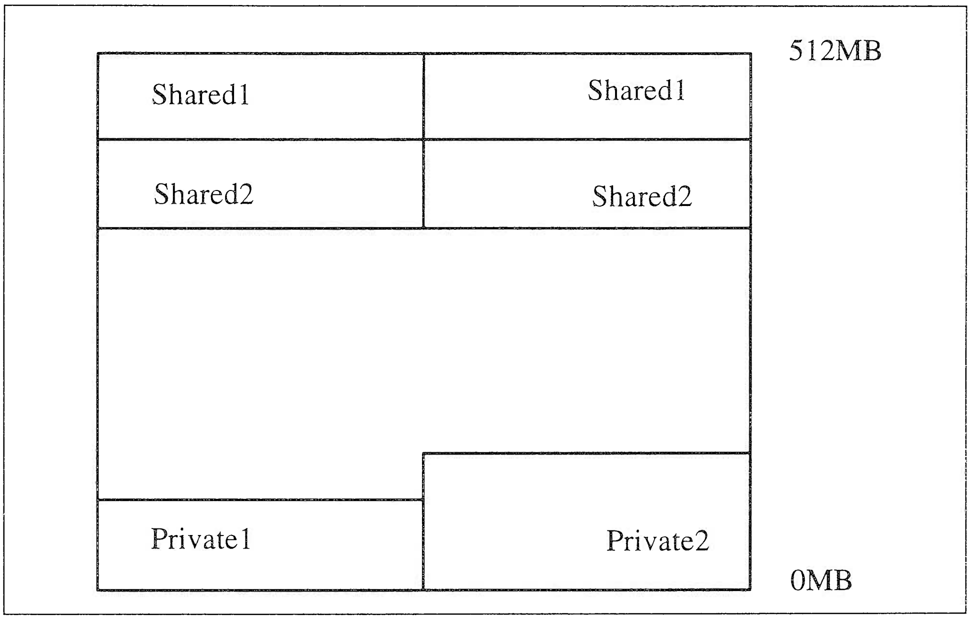


Figure 8-6. Private and shared arenas for two processes.

By using the DosAllocSharedMem API, instead of DosAllocMem, a program can allocate a shared memory object that can be used by other processes, even ones that exist in another session. As mentioned before, this shared memory object will reside in the shared arena of the process's virtual-address space. It will have the same physical address location across all processes.

DosAllocSharedMem

```
DosAllocSharedMem(
                        PPVOID BaseAddress,
                        PSZ    SharedMemName
                        ULONG  SharedMemSize
                        ULONG  AllocationFlags);
```

Parameters:

- ❑ BaseAddress (PPVOID) output—Essentially indicates the name of the variable to which you want the allocated shared memory assigned.
- ❑ SharedMemName (PSZ) input—Specifies the shared name of the memory object, which must begin with \SHAREMEM\ and must have the same format

as an OS/2 filename. For example, \SHAREMEM\SPAN.OBJ is a valid shared object name. A name need not be specified, but in such anonymous cases, another process cannot use the DosGetNamedSharedMem API to access the memory object without the proper memory allocation flags set.

- ❑ SharedMemSize (ULONG) input—Refers to the initial size of allocation. (As mentioned earlier, the number of bytes allocated will be rounded up to the next 4K page.)

- ❑ AllocationFlags (ULONG) input—Specifies the allocation attributes for the memory object. In addition to the flags that could have been specified with the DosAllocMem API, DosSetMem also allows the following:

Flag	**Value**	**Description**
OBJ_GETTABLE	(0x00000100)	Allows another process to get the memory object with the DosGetSharedMem API.
OBJ_GIVEABLE	(0x00000200)	Allows the memory object to be given to another process with the DosGiveSharedMem API.

These two flags can be specified only if the object was not given a name. If a shared memory object has these attributes, processes can use the corresponding APIs to access them. The DosSetMem API can also change the attributes of a shared memory object.

Figure 8-7, PROG8C.C, shows an example of allocating named shared memory. A corresponding program, such as PROG8C.C in Figure 8-8, can access that memory by using the DosGetNamedSharedMem API.

DosGetNamedSharedMem

```
DosGetNamedSharedMem(
                    PPVOID BaseAddress,
                    PSZ SharedMemName,
                    ULONG AttributeFlags);
```

Parameters:

- ❑ BaseAddress (PPVOID) output—Essentially indicates the name of the pointer variable to which you want the retrieved shared memory assigned.

- ❑ SharedMemName (PSZ) input—Specifies the name of the shared memory object that was allocated with the DosAllocSharedMem API.

- ❑ AttributeFlags (ULONG) input—Specifies the desired access protection for the memory object:

Flag	**Value**	**Description**
PAG_READ	(0x00000001)	Read access allowed.
PAG_WRITE	(0x00000002)	Write access allowed (also implies read and execute access).
PAG_EXECUTE	(0x00000004)	Execute access allowed (also implies read access).
PAG_GUARD	(0x00000008)	Guard page attribute is set.

At a minimum, the PAGE_READ, the PAG_WRITE, or the PAG_EXECUTE flag must be specified.

```
/* prog8c.c - Uses shared memory.                              */
/* (prog8d.c is the companion program.)                        */

#define INCL_DOSMEMMGR           /* Include Memory Management APIs */
#define INCL_DOSSEMAPHORES       /* Include Semaphore APIs         */
#define INCL_NOPMAPI             /* Don't include PM APIs          */
#include <os2.h>
#include <stdio.h>
#include <bsememf.h>             /* Flags for memory management    */

VOID main (USHORT argc, PCHAR argv[])
{
   PVOID    ObjAlloc;           /* Pointer to memory object        */
   ULONG    ObjSize;            /* Size of memory object           */
   ULONG    AllocFlags;         /* Attributes of the memory object */
   APIRET   waitrc,rc;          /* Return codes                    */
   UCHAR    SharedMemName[20];  /* Shared memory name              */
   UCHAR    SemEventName[20];   /* Semaphore created by prog8c.exe */
   UCHAR    SemEventName2[20]; /* Semaphore created by prog8ca.exe */
   HEV      hev, hev2;         /* Semaphore handles                */
   ULONG    flAttr = 0;        /* Semaphore flags = none           */
   BOOL32   fState = FALSE;    /* Semaphore initial state = set    */
```

```
int       i;                 /* Counter                            */

strcpy(SharedMemName,"\\SHAREMEM\\MEMEX.DAT");
ObjSize = 500;                       /* Will be rounded up to 4KB */

/* Can't specify the OBJ_GETTABLE flag since the memory           */
/* object is named; however, commit it and give it write access   */
AllocFlags = PAG_WRITE | PAG_COMMIT;

rc = DosAllocSharedMem(&ObjAlloc, SharedMemName, ObjSize,
        AllocFlags);

if (rc == 0) {
   printf("DosAllocSharedMem for %s executed successfully.\n",
     SharedMemName);
} else {
   printf("Error:  DosAllocSharedMem for %s with rc = %ld\n",
      SharedMemName,rc);
   return;
}

/* Name the two semaphores                                        */
strcpy(SemEventName,"\\SEM32\\MEMEX1");
strcpy(SemEventName2,"\\SEM32\\MEMEX2");
hev = 0;

/* Create the first semaphore                                     */
rc = DosCreateEventSem(SemEventName, &hev, flAttr, fState);

if (rc == 0) {
   printf("DosCreateEventSem for %s executed successfully.\n",
     SemEventName);
} else {
   printf("Error:  DosCreateEventSem for %s with rc = %ld\n",
     SemEventName,rc);
   return;
}

/* Fill the shared memory object with data                      */

strcpy(ObjAlloc,"A");
for (i=1;i<ObjSize;i++) {
   strcat(ObjAlloc,"A");
}
printf("ObjAlloc = %s\n",ObjAlloc);

/* Signal the first semaphore                                     */

DosPostEventSem(hev);
hev2 = 0;
do {
   /* Loop until semaphore 2 can be opened                        */
   rc = DosOpenEventSem(SemEventName2,&hev2);
```

```
      if (rc == 0) {
         printf("DosOpenEventSem for %s executed successfully.\n",
            SemEventName2);
      } else {
         printf("Error:  DosOpenEventSem for %s with rc =
            %ld\n",  SemEventName2,rc);
         printf("Still waiting...");
         DosSleep(1);
      }
   } while (rc != 0);
   printf("\n");

   /* Wait until semaphore 2 is signaled                             */
   DosWaitEventSem(hev2, SEM_INDEFINITE_WAIT);

   /* Close access to both semaphores                                */
   DosCloseEventSem(hev);
   DosCloseEventSem(hev2);

   /* Free the shared memory object                                  */
   rc = DosFreeMem(ObjAlloc);

   if (rc == 0) {
      printf("DosFreeMem for %s executed successfully.\n",
        SharedMemName);
   } else {
      printf("Error:  DosFreeMem for %s with rc = %ld\n",
        SharedMemName,rc);
   }

   return;
}
```

Figure 8-7. *DosAllocSharedMem (PROG8C.C).*

The related program in Figure 8-8 shows how to access memory by using the DosGetNamedSharedMem API.

```
/* prog8d.c - Uses shared memory.                                    */
/* (prog8c.c is the companion program.)                              */

#define INCL_DOSMEMMGR           /* Include Memory Management APIs */
#define INCL_DOSSEMAPHORES       /* Include Semaphore APIs         */
#define INCL_NOPMAPI             /* Don't include PM APIs          */
#include <os2.h>
#include <stdio.h>
#include <bsememf.h>                /* Flags for memory management */

VOID main (USHORT argc, PCHAR argv[])
{
   PVOID  ObjAlloc;             /* Pointer to memory object        */
   ULONG  ObjSize;              /* Size of memory objects          */
ULONG   AllocFlags;            /* Attributes of the memory objects */
APIRET rc;                     /* Return code                      */
UCHAR   SharedMemName[20];     /* Shared memory name               */
UCHAR   SharedEventName[20];   /* Semaphore created by prog8c.exe  */
UCHAR   SharedEventName2[20];  /* Semaphore created by prog8ca.exe */
HEV     hev, hev2;             /* Semaphore handles                */
ULONG   flAttr = 0;            /* Semaphore flags = none           */
BOOL32 fState = FALSE;         /* Semaphore initial state = set    */

/* Initialize names                                                */
strcpy(SharedMemName,"\\SHAREMEM\\MEMEX.DAT");
strcpy(SemEventName,"\\SEM32\\MEMEX1");
strcpy(SemEventName2,"\\SEM32\\MEMEX2");
hev = 0;
hev2 = 0;

rc = DosCreateEventSem(SemEventName2, &hev2, flAttr, fState);

if (rc == 0) {
   printf("DosCreateEventSem for %s executed successfully.\n",
     SemEventName2);
} else {
   printf("Error:  DosCreateEventSem for %s with rc = %ld\n",rc);
   return;
}

/* Try to open the semaphore created by the first program         */
rc = DosOpenEventSem(SemEventName,&hev);
if (rc == 0) {
   printf("DosOpenEventSem for %s executed successfully.\n",
     SemEventName);
} else {
   /* This program expects the first semaphore to exist.          */
   /* You could set it up to loop until it does.                  */
   printf("Error:  DosOpenEventSem for %s with rc = %ld\n",
     SemEventName,rc);
   DosCloseEventSem(hev2);
   return;
}

/* Wait for the first semaphore to be posted                       */
DosWaitEventSem(hev, SEM_INDEFINITE_WAIT);

AllocFlags = PAG_WRITE;

rc = DosGetNamedSharedMem(&ObjAlloc,SharedMemName,AllocFlags);
if (rc == 0) {
   printf("DosGetNamedSharedMem for %s executed
      successfully.\n", SharedMemName);
} else {
   printf("Error:  DosGetNamedSharedMem for %s
     with rc = %ld\n", SharedMemName,rc);
}
```

```
   printf("ObjAlloc = %s\n",ObjAlloc);

   rc = DosFreeMem(ObjAlloc);

   if (rc == 0) {
      printf("DosFreeMem for %s executed successfully.\n",
        SharedMemName);
   } else {
      printf("Error:  DosFreeMem for %s with rc = %ld\n",
        SharedMemName,rc);
   }

   DosPostEventSem(hev2);

   /* Pause to let the other program catch the semaphore before     */
   /* closing it.                                                   */
   printf("Pausing...\n");
   DosSleep(2);

   /* Now close the semaphores and leave                            */
   DosCloseEventSem(hev);
   DosCloseEventSem(hev2);

   return;
}
```

Figure 8-8. *DosGetSharedMem (PROG8D.C).*

PROG8C.C in Figure 8-7 first allocates a 500-byte memory object, which is named \SHAREMEM\MEMEX.DAT. Afterwards, it creates semaphore \SEM32\MEMEX1, fills the shared memory object, and waits for PROG8D.C in Figure 8-8 to post semaphore \SEM32\MEMEX2.

After PROG8D.C in Figure 8-8 waits for PROG8C.C in Figure 8-7 to post semaphore \SEM32\MEMEX1, the program reads in the memory object, posts semaphore \SEM32\MEMEX2, and cleans up its objects before terminating.

When signaled, PROG8C.C in Figure 8-7 frees and closes its objects before ending. The system finally frees the shared memory when its usage count goes to zero. This will occur after PROG8C.C in Figure 8-7 frees it. Figure 8-10 on the next page shows the flow.

To run the example in Figure 8-10, first execute PROG8C.C in one session, such as an OS/2 windowed session. As PROG8C.C loops, waiting for the second semaphore to be created, run PROG8D.C in another OS/2 windowed session. Both programs will play out, printing the contents of the shared memory object and status messages before ending.

The output for PROG8C.C is shown in Figure 8-9:

```
DosAllocSharedMem for \SHAREMEM\MEMEX.DAT executed successfully.
DosCreateEventSem for \SEM32\MEMEX1 executed successfully.
ObjAlloc = AAAAAAAAA....AAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Error:  DosOpenEventSem for \SEM32\MEMEX2 with rc = 187
Still waiting........................
..................................................
DosOpenEventSem for \SEM32\MEMEX2 executed successfully.
DosFreeMem for \SHAREMEM\MEMEX.DAT executed successfully.
```

***Figure 8-9**. Output of PROG8C.C.*

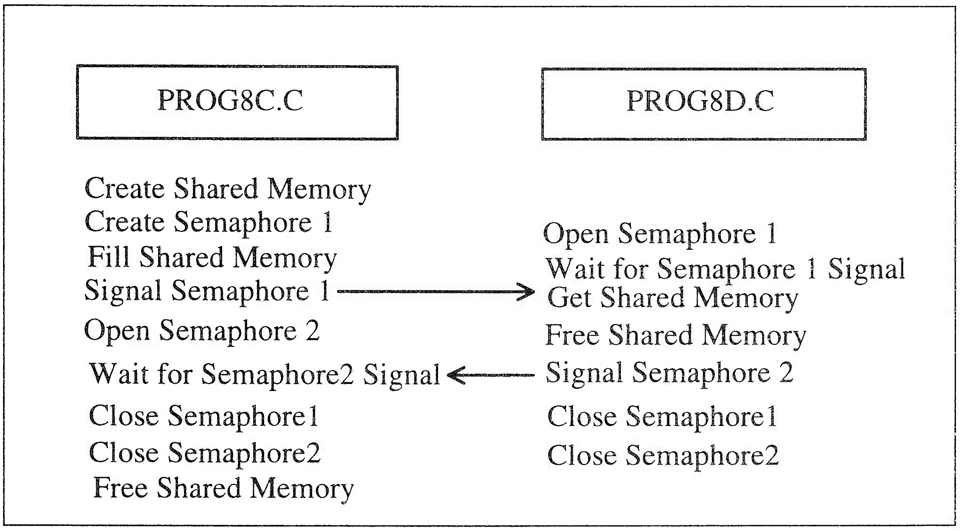


***Figure 8-10.** The flow of PROG8C.C and PROG8D.C shared memory use.*

The output for PROG8D.C is shown in Figure 8-11 on the next page.

```
DosCreateEventSem for \SEM32\MEMEX2 executed successfully.
DosOpenEventSem for \SEM32\MEMEX1 executed successfully.
DosGetNamedSharedMem for \SHAREMEM\MEMEX.DAT executed successfully.
ObjAlloc = AAAAAAAAAAAA......AAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
DosFreeMem for \SHAREMEM\MEMEX.DAT executed successfully.
```

Figure 8-11. *Output for PROG8D.C 9 in Figure 8-8.*

Both PROG8C.C and PROG8D.C used semaphores to wait for each other to complete tasks. Another Interprocess Communication (IPC) object could have been used for signalling instead. Also, not all the return codes for called APIs were checked in the programs; they could have been.

Note that you can also use the DosSetMem API to change the attribute or desired access protection flags for shared memory objects. However, you cannot decommit the pages of a shared memory object for obvious reasons.

MEMORY POOLS

After memory is allocated with the DosAllocMem API, it can be initialized as a memory pool for suballocation with the DosSubSetMem API. Using pool memory will require additional application logic that the DosAllocMem API does not.

This feature can be useful if an application needs to allocate many small objects and does not want to incur the overhead of a minimum allocation of one page per object. Thus, it also reduces the chances for widespread memory fragmentation.

DosSubSetMem

```
DosSubSetMem(
             PVOID BaseAddress,
             ULONG Allocation,
             ULONG PoolSize);
```

Parameters:

- ❑ BaseAddress (PVOID) input—Specifies the same variable name, essentially the address, that was previously allocated with the DosAllocMem API.

- ❑ Allocation (ULONG) input—Allows the following allocation (or execution) attributes:

Attribute	**Value**	**Description**
DOSSUB_INIT	(0x00000001)	Initialize a memory object as a pool for suballocation.
DOSSUB_GROW	(0x00000002)	Indicate that the size of a previously initialized object should be increased, as specified by the PoolSize parameter.
DOSSUB_SPARSE_OBJ	(0x00000004)	Direct the suballocation functions to manage the page commitment in the memory pool. If specified, all pages in the memory object must not have been already committed. The converse is true if this flag is not specified.
DOSSUB_SERIALIZE	(0x00000008)	Indicate that access to the pool should be serialized. If the DosSubSetMem API is called a second time, such as to increase a previously subsetted amount of memory, the DOSSUB_SPARSE_OBJ and DOSSUB_SERIALIZE flags must be set if they were during the first call. Note that all pages have the same attributes with read and write access being the minimum specified. DosSetMem should not be called to change the attributes of the pages for the memory object lest the program encounter unpredictable results.

- ❑ PoolSize (ULONG) input—Indicates the size of the pool to subset. This value can be the same as was specified by the DosAllocMem API.

Increasing a memory pool has certain advantages. If the initial allocation with DosAllocMem did not commit the pages, the actual memory for the object would not have been allocated. Thus, it doesn't take space in physical memory, whether in RAM or in the swap file.

By using the DOSSUB_SPARSE_OBJ flag, memory commitment is handled by the DosSubAllocMem API, and memory will get allocated when it is needed.
After initializing a memory pool with the DosSubSetMem API, subobjects can be allocated from within the pool by using the DosSubAllocMem API

DosSubAllocMem

```
DosSubAllocMem(
               PVOID BaseAddress,
               PPVOID BlockOffset,
               ULONG Size);
```

Parameters:

- ❑ BaseAddress (PVOID) input—Specifies the same value as the memory pool object variable name that was allocated with the DosSubSetMem API.
- ❑ BlockOffset (PPVOID) output—Specifies the memory subobject to allocate.
- ❑ Size (ULONG) input—Indicates the byte size of the subobject. If not in a multiple of 8 bytes, the size will be rounded up to the nearest 8 bytes.

Lastly the DosSubUnsetMem API uninitializes the memory pool. To free the entire pool memory back to the system, the DosFreeMem API must still be called.

DosSubUnsetMem

```
DosSubUnsetMem(
               PVOID BaseAddress);
```

Parameter:

- ❑ BaseAddress (PVOID) input—Specifies the same address as the memory pool object variable name that was allocated with the DosSubSetMem API.

PROG8E.C in Figure 8-12 shows an example of using a memory pool. The pool is first initialized to 1000 bytes out of the 4000 bytes available to the object. The program carves out one subobject of 500 bytes from the pool, but it cannot carve out the second subobject of 1000 bytes until it increases the pool size. Afterwards, the subobjects, pool, and base memory objects are freed. The output for PROG8E.C is shown in Figure 8-13.

```
/* prog8e.c - Demonstrate pool memory.                              */

#define INCL_DOSMEMMGR           /* Include Memory Management Calls */
#define INCL_DOSERRORS           /* Include OS/2 Error #defines     */
#define INCL_NOPMAPI             /* Don't include PM APIs           */
#include <os2.h>
#include <stdio.h>
#include <bsememf.h>             /* Flags for memory management     */

VOID main (USHORT argc, PCHAR argv[])
{
   PVOID  PoolAlloc;             /* Pointer to pool memory object   */
   PVOID  ObjAlloc1, ObjAlloc2;  /* Pointer to  subobjects          */
                                 /* in the pool memory object       */
   ULONG  PoolSize;              /* Size of the pool memory object  */
   ULONG  ObjSize1, ObjSize2;    /* Size of the memory subobjects   */
                              /* Attributes of the memory subobjects */
   ULONG  AllocFlags1, AllocFlags2;
   APIRET rc;                    /* Return code                     */
   int i;                        /* Counter                         */

   /* Will be rounded up to the nearest 8 bytes - to 4008           */
   PoolSize = 4002;
   ObjSize1 = 500;               /* Size of subobject 1             */
   ObjSize2 = 1000;              /* Size of subobject 2             */

   /* Precommit the pages                                           */
   AllocFlags1 = PAG_WRITE | PAG_READ | PAG_COMMIT;

   rc = DosAllocMem(&PoolAlloc, PoolSize, AllocFlags1);

   if (rc == 0) {
      printf("DosAllocMem for PoolAlloc executed successfully.\n");
   } else {
      printf("Error:  DosAllocMem for PoolAlloc with rc = %ld\n",rc);
      return;
   }

   AllocFlags2 = DOSSUB_INIT;

   /* Only allocate 1000 bytes of the memory object as an           */
   /* available pool                                                */
   rc = DosSubSetMem(PoolAlloc, AllocFlags2, (PoolSize - 3000));
   if (rc == 0) {
      printf("DosSubSetMem for PoolAlloc executed successfully.\n");
   } else {
      printf("Error:  DosSubSetMem for PoolAlloc with rc = %ld\n",
         rc);
      return;
   }
```

```
rc = DosSubAllocMem(PoolAlloc, &ObjAlloc1, ObjSize1);

if (rc == 0) {
   printf("DosSubAllocMem for ObjAlloc1 executed successfully.\n");
} else {
   printf("Error:  DosSubAllocMem for ObjAlloc1 with rc = %ld\n",
      rc);
   DosFreeMem(PoolAlloc);
   return;
}

/* Fill subobject 1                                                  */
strcpy(ObjAlloc1,"A");
for (i=1;i<ObjSize1;i++) {
   strcat(ObjAlloc1,"A");
}
printf("ObjAlloc1 = %s\n",ObjAlloc1);

rc = DosSubAllocMem(PoolAlloc, &ObjAlloc2, ObjSize2);

if (rc == 0) {
   printf("DosSubAllocMem for ObjAlloc2 executed successfully.\n");
} else {
   printf("Error:  DosSubAllocMem for ObjAlloc2 with rc = %ld\n",
      rc);

   /* Check if the error was not enough room in the pool             */
   if (rc == ERROR_DOSSUB_NOMEM) {
      printf("Increasing size of the memory pool.\n");
      AllocFlags2 = DOSSUB_GROW ;
      rc = DosSubSetMem(PoolAlloc, AllocFlags2, 2000);
      if (rc == 0) {
         /* Allocate memory for subobject 2                          */
         printf("DosSubSetMem executed successfully.\n");
         DosSubAllocMem(PoolAlloc, &ObjAlloc2, ObjSize2);
      } else {
         printf("DosSubSetMem error: return code = %ld\n",rc);
         printf("Exiting program.\n");
         return;
      }

   }
}

/* Fill subobject 2                                                  */
strcpy(ObjAlloc2,"B");
for (i=1;i<ObjSize2;i++) {
   strcat(ObjAlloc2,"B");
}
printf("ObjAlloc2 = %s\n",ObjAlloc2);

/* Free subobjects                                                   */
DosSubFreeMem(PoolAlloc, ObjAlloc1, ObjSize1);
DosSubFreeMem(PoolAlloc, ObjAlloc2, ObjSize2);
```

```
    /* Free Memory Pool                                             */
    rc = DosSubUnsetMem(PoolAlloc);

    if (rc == 0) {
       printf("DosSubUnsetMem for PoolAlloc executed successfully.\n");
    } else {
       printf("Error:  DosSubUnsetMem for PoolAlloc with rc = %ld\n",
          rc);
    }

    /* Free Memory Pool Allocation                                  */
    rc = DosFreeMem(PoolAlloc);

    if (rc == 0) {
       printf("DosFreeMem for PoolAlloc executed successfully.\n");
    } else {
       printf("Error:  DosFreeMem for PoolAlloc with rc = %ld\n",rc);
    }

    return;
}
```

Figure 8-12. *DosSubAllocMem and DosSubSetMem (PROG8E.C).*

```
DosAllocMem for PoolAlloc executed successfully.
DosSubSetMem for PoolAlloc executed successfully.
DosSubAllocMem for ObjAlloc1 executed successfully.
ObjAlloc1 = AAAAAAAAAAAAA.......AAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Error:  DosSubAllocMem for ObjAlloc2 with rc = 311
Increasing size of the memory pool.
DosSubSetMem executed successfully.
ObjAlloc2 = BBBBBBBBBBBBBBBBB.......BBBBBBBBBBB
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
DosSubUnsetMem for PoolAlloc executed successfully.
DosFreeMem for PoolAlloc executed successfully.
```

Figure 8-13. *Output for PROG8E.C in Figure 8-12.*

SWAPPING

When an application exceeds the size of physical memory on a workstation, all or part of it will be swapped to disk. This swapping process is transparent to the application. The OS/2 2.1 swap file is named SWAPPER.DAT. It is located in the path identified by the SWAPPATH statement in the CONFIG.SYS file. This statement also has parameters to indicate the initial size of the swap path and the increments by which the file should grow. For example, the following statement indicates that the full path name of the swap path file will be D:\OS2\SYSTEM\SWAPPER.DAT. It will be initialized to 2MB and grow or shrink in 1MB increments as needed thereafter.

```
SWAPPATH=D:\OS2\SYSTEM 2048 1024
```

The swapper file can be placed anywhere on a hard disk. Placing it in a partition of its own will help reduce the fragmentation of the file. If the partition is an HPFS drive, as opposed to FAT, this will also increase the efficiency at which pages can be swapped to and from the file. (A similar benefit holds true if the swap file resides on an HPFS partition with other files.) Note that if the swapper cannot grow because it has run out of space, for example, the user must free up additional hard disk space or close running applications.

OS/2 1.x would swap an entire object to the swap file, which proved complicated since an object could be of variable length. OS/2 swaps pages with fixed sizes (4KB) to and from the file. This simplifies the process greatly. In addition, OS/2 2.1 does not require contiguous pages for most memory objects, so compaction of the swap file is not as essential as it was in OS/2 1.x.

Pages that have not been accessed for a period of time are swapped to disk when additional physical memory is needed. If a memory object is located in a page in the swap file and a program running in physical memory references it, the page will be swapped in and another swapped out, if necessary.

EXEs, DLLs, and other memory objects can also be moved to the swap file when other processes need physical memory in the system. The swapper file can shrink, too, but only if all the pages in a growth increment of the file are no longer referenced. This happens infrequently. As previously mentioned, the swap file growth increment was defined in the SWAPPATH statement.
If for some reason the user does not want memory swapped, the MEMMAN statement can be changed in the CONFIG.SYS file. Upon installation this statement is usually set to the following, which allows swapping and protected memory:

```
MEMMAN=SWAP,PROTECT
```

Specifying NOSWAP would prevent swapping to disk. Unfortunately, it would also inhibit the system from exceeding physical memory. If PROTECT was not specified, application programs would not be able to allocate protected memory. The NOLOAD parameter also exists, but it was carried over for OS/2 1.x compatibility. It has no bearing in OS/2 2.1.

If a system has little physical memory, the workstation can start thrashing, meaning that it must frequently, perhaps constantly, swap memory to and from the swap file. Thrashing will severely degrade system performance if you don't take appropriate steps.

Just as in the case where the swap file cannot grow, to alleviate thrashing you can reduce the number of processes actively running in the system, tune buffer or swap parameters, and install additional memory on the workstation.

API SUMMARY

Memory management APIs, both those used in examples in this chapter and those that weren't, are listed below:

- ❑ DosAllocMem—Allocate a private memory object.
- ❑ DosAllocSharedMem—Allocate a shared memory object.
- ❑ DosFreeMem—Free a private or shared memory object.
- ❑ DosGetNamedSharedMem—Gain access to a named shared memory object.
- ❑ DosGiveSharedMem—Provide access to a shared memory object.
- ❑ DosQueryMem—Retrieve attribute information on all or part of a memory object.
- ❑ DosSetMem—Change the attributes or commit state for a memory object and its pages.
- ❑ DosSubAllocMem—Allocate a memory subobject from a pool memory.
- ❑ DosSubFreeMem—Free a previously allocated subobject from pool memory.

- ❑ DosSubSetMem—Initialize an allocated memory object for pool memory.
- ❑ DosSubUnsetMem—Uninitialize an allocated memory object for pool memory.

FLAG SUMMARY

Table 8-1 on lists flags used by memory APIs that are not specific to pools.

Value	Name \ APIs	1	2	3	4	5
0x00000001	PAG_READ	X	X	X	X	X
0x00000002	PAG_WRITE	X	X	X	X	X
0x00000004	PAG_EXECUTE	X	X	X	X	X
0x00000008	PAG_GUARD	X	X	X	X	X
0x00000010	PAG_COMMIT	X	X	X	X	
0x00000040	OBJ_TILE	X			X	
0x00000020	PAG_DECOMMIT		X			
0x00000100	OBJ_GETTABLE				X	
0x00000200	OBJ_GIVEABLE				X	
0x00000400	PAG_DEFAULT		X			
0x00002000	PAG_SHARED			X		
0x00004000	PAG_FREE			X		
0x00010000	PAG_BASE			X		

The numbers in the heading represent the following APIs:

1 DosAllocMem—PAG_READ, PAG_WRITE, or PAG_EXECUTE must be set, at a minimum.

2 DosSetMem—At a minimum, PAG_READ, PAG_WRITE, PAG_EXECUTE, PAG_DEFAULT, or PAG_DECOMMIT must be set.

3 DosQueryMem—Flags are returned to this API as output and are used for informational purposes.

4 DosAllocSharedMem—At a minimum, PAG_READ, PAG_WRITE, or PAG_EXECUTE must be set.

5 DosGetNamedSharedMem—At a minimum, PAG_READ, PAG_WRITE, or PAG_EXECUTE must be set.

Table 8-1. *Flags for memory APIs not specific to pools.*

Table 8-2 lists flags used by the memory API that are specific to pools, as allocated with the DosSubSetMem API:

Value	Name
0x00000001	DOSSUB_INIT
0x00000002	DOSSUB_GROW
0x00000004	DOSSUB_SPARSE_OBJ
0x00000008	DOSSUB_SERIALIZE

Table 8-2. *Flags used by the memory API that are specific to pools.*

TIPS

The following are tips to keep in mind when dealing with memory management in your programs:

- ❑ You may want to use the malloc and free C library functions sometimes. Internally, they will call the DosAllocMem and DosFreeMem APIs. For greater abstraction, use these functions. For greater control over your memory objects, use the OS/2 APIs.

- ❑ If memory usage is a concern, don't allocate a lot of small objects (< 4KB) with the DosAllocMem API, since they will consume the same number of pages as the number of objects. This will waste a high proportion of space. Use pool memory or another construct, such as a C language compound structure, to pack your memory objects closer together. Depending on the implementation of your compiler, using the C library malloc call could also reduce fragmentation.

- ❑ It's easier to commit all the pages for your memory objects upon allocation. However, if memory usage is a concern, stage your commits and grow the amount of memory needed for an object upon demand. You can take advantage of such constructs as the guard page to flag when more memory is needed.

- ❑ Shared memory names are easier to use than unnamed shared memory. This works only if the process knows the name beforehand or can get to it easily.

- ❑ Put the swap file in a partition of its own, whether FAT or HPFS, to reduce fragmentation.

- ❑ Be sure to use DosFreeMem if you used DosAllocMem to create a memory object. Correspondingly, use the C library free function if you allocated memory with the malloc C library function. You will cause a system trap if you use the free function for an object created with DosAllocMem. This can occur if a DLL allocated memory with DosAllocMem and the calling application has the responsibility to free the allocated memory. In this case, the application program should use DosFreeMem and not the free function to return memory to the system. Using a consistent strategy will save some debugging time in the long run.

- ❑ Freeing memory in a procedure call is especially important for functions within a DLL. If temporary memory isn't freed, OS/2 will not be able to recover the memory until the EXE owning the DLL ends. This is how memory "leaks" happen.

SUMMARY

OS/2 2.1 will lift the heavy burden of memory management from your shoulders. Now you can to devote your energies to solving more user-oriented tasks, such as application functionality and user friendliness. OS/2 2.1 accomplishes memory management with the use of the flat (or linear) memory model. The use of fixed-sized pages eliminates the need for variable-sized segmented objects. You have the flexibility to specify the attributes of pages and defer their commitment to physical memory addresses until a later time.

You can also take advantage of pool memory to reduce the fragmentation of memory objects. The process memory for an application is divided into its private and shared arenas. This provides protection to a single process and also facilitates sharing across processes. Whether a process uses EXEs, DLLs, private, or shared memory objects, OS/2 2.1 uses a swap file to allow your application to exceed the size of physical memory on the workstation by large degrees.

CHAPTER 9

Multitasking

"We have before us the opportunity to forge for ourselves and for future generations a new world order, a world where the rule of law, not the law of the jungle, governs the conduct of nations." —*President George Bush*

INTRODUCTION

From the start, OS/2 was designed as a multitasking operating system. DOS, on the other hand, is single-tasking. It is ill-prepared to run more than one program concurrently. With the OS/2 2.1 multitasking environment, the ability to run multiple programs (or processes) is embedded into the operating system kernel. Without this built-in support, applications would be forced to maintain their own code to divide work. Programs would have to internally juggle printing and calculating while trying to be responsive to user requests at the same time.

To help overcome this deficiency in DOS, programs (and programmers too) have to rely on DOS extensions which only multitask to a limited degree. These extenders provide cooperative multitasking, which means programs must specifically code when they can relinquish control of the CPU. Only then can the CPU turn its attention to another waiting task. OS/2 2.1 leaves these patches and extensions behind.

Instead, OS/2 2.1 treats hardware as a shared resource that multiple programs can use. Hardware is not shackled to the exclusive use of one process. OS/2 also removes the burden of multitasking from applications by switching between tasks in a preemptive fashion; process tasks are given slices of CPU time in which to execute. In a preemptive environment, a higher priority task can interrupt the CPU and prevent the lower priority task that was running from completing its CPU timeslice. The frequency and duration of these timeslices depends on the process's priority, the number of interrupts that the processor receives, and the presence of another, higher priority process barging onto the scheduling scene. In fact, processes themselves are subdivided into smaller units of work, called threads. The hierarchy of multitasking, which includes sessions, is discussed later.

PROTECTION

OS/2 also provides a high degree of protection to application programs running in its multitasking environment. Each process has its own virtual address space among other resources. If a process should go down from an error, it brings down only the session (in which the process was running), rather than fouling the entire operating system. Some DOS applications have the potential of doing this in DOS. This degree of crash protection can be graphically illustrated in OS/2 2.1 by compiling a DOS program that divides by zero and running it in a DOS session.

The program will quickly crash and summon the trap screen. After returning the error to the program, it and its session are removed from the OS/2 desktop and process hierarchy. The DOS program need not divide by zero; accessing an element outside of a defined array will produce the same regrettable results.

Try executing the same program under a DOS or DOS extender environment and see if your machine is still accessible afterwards.

SESSIONS

The granularity of multitasking goes beyond the program level. OS/2 2.1 breaks multitasking groups into sessions, processes, and threads. This hierarchy is shown in Figure 9-1. Sessions head the top of the hierarchy. They represent virtual machines.

Each session has a logical keyboard, mouse, screen, and I/O devices. These logical devices map to real ones when the session is brought to the foreground in the case of the keyboard and mouse. Only one session at a time can execute in the foreground; all others run in the background. For example, when you move your mouse pointer around the OS/2 2.1 desktop and click on a window or application, that object is brought not only to the top of the display but to the foreground as well.

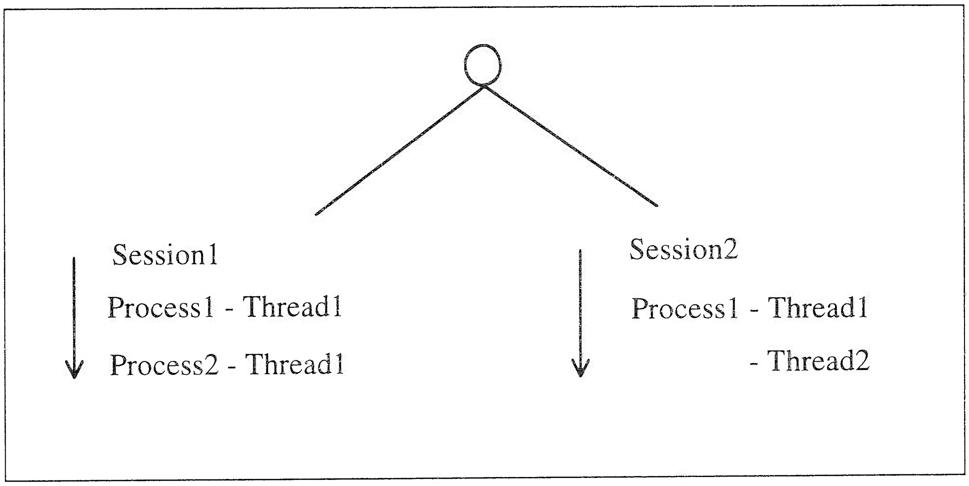


Figure 9-1. *Multitasking hierarchy.*

Some of the OS/2 2.1 windows on the desktop screen exist as sessions themselves while others are part of a single session. Applications such as the OS/2 2.1 Enhanced Editor execute in their own session. Presentation Manager (PM) runs its own session too. OS/2 2.1 supports up to 255 sessions at a time. Each session, process, and thread has a unique ID to identify them in the process tree.

The first code example, shown in Figure 9-2, demonstrates the use of the DosStartSession API to execute a program in another, newly created session.

DosStartSession

```
DosStartSession(
                    PSTARTDATA SessionDataStruct,
                    PULONG SessionID,
                    PPID ProcessID);
```

Parameters:

- ❑ SessionDataStruct (PSTARTDATA) input—Created session's data structure. Before starting the session, the program must first initialize the STARTDATA data structure, which is discussed in further detail below.

- ❑ SessionID (PULONG) output—Returned ID of the created session, if related. You can specify whether a session is related to or independent of the spawning session in the STARTDATA structure.

- ❑ ProcessID (PPID) output—Returned ID of the created process in the session, if started as a related session.

The STARTDATA Structure

This section briefly discusses the STARTDATA structure. For more information on this structure, the DosStartSession API, or any of the APIs in this book, consult your OS/2 2.1 Control Program Programming Reference documentation.

```
typedef struct _STARTDATA
        {
         USHORT  Length;
         USHORT  Related;
         USHORT  FgBg;
         USHORT  TraceOpt;
         PSZ     PgmTitle;
         PSZ     PgmName;
         PBYTE   PgmInputs;
         PBYTE   TermQ;
         PBYTE   Environment;
         USHORT  InheritOpt;
         USHORT  SessionType;
         PSZ     IconFile;
         ULONG   PgmHandle;
         USHORT  PgmControl;
         USHORT  InitXPos;
         USHORT  InitYPos;
         USHORT  InitXSize;
         USHORT  InitYSize;
         USHORT  Reserved;
         PSZ     ObjectBuffer;
         ULONG   ObjectBuffLen;
        } STARTDATA;
```

Parameters:

- ❑ Length—Byte size of the structure: 24, 30, 32, 50, or 60 bytes. If under 32, DosStartSession fills in values that may not be present, such as the process's installation program. Lengths over 32 bytes are allowed only if PM is present. As the size of the structure increases, more definition and inheritance options may be defined and used. If you want to specify the buffer used to store the executed process's error message, you'll want all 60 bytes.

- Related—The created session may be started as an independent or as a child session. Independent sessions cannot be controlled by the session that created it, nor can its termination queue be accessed. The API will not return the created session ID or its process ID for independent session. Possible values are:

```
0   SSF_RELATED_INDEPENDENT
1   SSF_RELATED_CHILD
```

- FgBg—Starts the session in the foreground, where it gains focus, or in the background. Possible values are:

```
0   SSF_FGBG_FORE
1   SSF_FGBG_BACK
```

- TraceOpt—Indicates whether the session should include tracing options. Possible values are:

```
0   SSF_TRACEOPT_NONE
1   SSF_TRACEOPT_TRACE
2   SSF_TRACEOPT_TRACEALL (which includes descendents)
```

- PgmTitle—Title of the program; defaults to the program name if this parameter is not specified.

- PgmName—Fully qualified name of the program to be executed in the session.

- PgmInputs—Arguments to the program.

- TermQ—Address of a system queue created before the API is called. Set this parameter to 0 if a queue is not defined. This termination queue is important if you want to retrieve the return code from the process executed in the session when it completes.

- Environment—Environment string to be passed to the program in the session. If set to zero, it inherits the environment according to the InheritOpt parameter.

- InheritOpt—Indicates the environment that the program in the session should inherit. Possible values are:

```
0  SSF_INHERTOPT_SHELL
1  SSF_INHERTOPT_PARENT (process making the API call)
```

  For example, set the Environment parameter to 0 and then set the InheritOpt parameter to SSF_INHJERITOP_SHELL if you want your starting session to inherit the environment of your shell.

- SessionType—Indicates the type of session that should be created in which to run the program. Possible values are:

```
0  SSF_TYPE_DEFAULT (default type taken from the PgmHandle)
1  SSF_TYPE_FULLSCREEN (full-screen OS/2 session)
2  SSF_TYPE_WINDOWABLEVIO (windowed OS/2 session with VIO)
3  SSF_TYPE_PM (windowed OS/2 session with PM and VIO)
4  SSF_TYPE_VDM (full-screen DOS session)
7  SSF_TYPE_WINDOWEDVDM (windowed DOS session)
```

- IconFile—Fully qualified icon file name. If set to 0, the default icon will be used.

- PgmHandle—Handle of the program. If set to 0, the program will use values in the STARTDATA structure for such specifications as the window size and position.

- PgmControl—State for the session to start in. This value and the x and y position and size values are used only for PM and VIO window applications. Possible values are:

```
0x0000  SSF_CONTROL_VISIBLE
0x0001  SSF_CONTROL_INVISIBLE
0x0002  SSF_CONTROL_MAXIMIZE
0x0004  SSF_CONTROL_MINIMIZE
0x0008  SSF_CONTROL_NOAUTOCLOSE (for VIO window apps)
0x8000  SSF_CONTROL_SETPOS (used specified size and position)
```

- InitXPos—Initial x coordinate.

- InitYPos—Initial y coordinate.

- ❑ InitXSize—Initial width.
- ❑ InitYSize—Initial height.
- ❑ Reserved—Set to zero. Saved for future use.
- ❑ ObjectBuffer—Buffer of the error message from the program that was executed in the session. This is similar to the one used by the DosExecPgm API.
- ❑ ObjectBuffLen—Byte length of the ObjectBuffer.

The session created by the PROG9A.C program in Figure 9-2 will run the command line shell program, CMD.EXE, with the directory command (DIR) as its parameter. The output will be directed to a file (DIR.OUT), shown in Figure 9-3. This file can be examined after the program executes.

Note that in DOS environments, the shell program is called COMMAND.COM.

The additional /C parameter for the CMD.EXE tells the command shell to terminate the session when the process ends. If /K was used, the session would remain in existence after the process in the session ends. Assuming the session was related to its creator, the child session must then be terminated with the DosStopSession API.

Sessions are valuable in scenarios where you are calling a command line program from a PM application. Since PM uses different logical devices than standard input and output (stdin and stdout), the DosExecPgm API cannot be used for these programs. You will need to start another related session and run the command line program in that new session.

If you need the return code from the command line program, create a termination queue and wait for an item to be posted on the queue. Read the 8-byte (two word) item from the queue and examine the second word of the item for the return code of the completed process.

Note that the first word has the session ID of the child. It can be used to check that the item on the queue belongs to the child session that was created. Be sure to free the item and the queue afterwards.

```
/*  Prog9a.c.   Creating a new OS/2 session.                       */

#define INCL_DOSSESMGR              /* Include Session Manager Calls */
#include <os2.h>
#include <stdio.h>

VOID main (void)
{
   STARTDATA   StartData;           /* Start Session Data Structure */
   ULONG       SessID;              /* Returned Session ID          */
   PID         PID;                 /* Returned Process ID          */
   UCHAR       PgmTitle[40];
   UCHAR       PgmInputs[40];
   UCHAR       PgmName[80];
   UCHAR       ObjBuf[100];
   APIRET      rc;

   /*  Specify the various session start parameters                 */
   /*  printf("Starting program.\n");                               */
   /*  Fill out the Session Data Structure                          */

   /* structure size                                                */
   StartData.Length = sizeof(STARTDATA);

   /* Independent Session                                           */
   StartData.Related = SSF_RELATED_INDEPENDENT;

   /* Start in background                                           */
   StartData.FgBg = SSF_FGBG_BACK;

   /* Do not include trace information                              */
   StartData.TraceOpt = SSF_TRACEOPT_NONE;

   /* Title of Program in Session                                   */
   strcpy(PgmTitle,"Command Line Program");
   StartData.PgmTitle = PgmTitle;

   /* Name of program to be executed in session                     */
   strcpy(PgmName,"CMD.EXE");
   StartData.PgmName = PgmName;

   /* Arguments to pass to program                                  */
   strcpy(PgmInputs,"/C DIR > \"DIR.OUT\"");
   StartData.PgmInputs = PgmInputs;

   /* No termination queue                                          */
   StartData.TermQ = 0;

   /* No environment string                                         */
   StartData.Environment = 0;

   /* Inherit environment from parent                               */
   StartData.InheritOpt = SSF_INHERTOPT_PARENT;
```

```
    /* Use the shell's default session type                    */
    StartData.SessionType = SSF_TYPE_DEFAULT;

    /* No icon file                                            */
    StartData.IconFile = 0;

    /* No Program handle                                       */
    StartData.PgmHandle = 0;

    /* Program starts invisible and minimized                  */
    StartData.PgmControl = SSF_CONTROL_INVISIBLE |
       SSF_CONTROL_MINIMIZE;
    StartData.InitXPos = 0;
    StartData.InitYPos = 0;
    StartData.InitXSize = 0;
    StartData.InitYSize = 0;

    /* Reserved word set to zero                               */
    StartData.Reserved = 0;

    /* Holds DosExecPgm failure message                        */
    StartData.ObjectBuffer = ObjBuf;
    StartData.ObjectBuffLen = sizeof(ObjBuf);

    printf("Starting the session.\n");

    rc = DosStartSession(&StartData, &SessID, &PID);

    if (rc != 0) {
       printf("DosStartSession error: return code = %ld\n", rc);
       printf("Object Buffer = %s\n", StartData.ObjectBuffer);
    } else {
       printf("DosStartSession executed successfully.\n");
       printf("Created Session ID = %ld\n", SessID);
       printf("Created Process ID in session = %ld\n", PID);
    }
    return;
}
```

Figure 9-2. *Creating a new OS/2 session. (PROG9A.C).*

```
Starting program.
Starting the session.
DosStartSession executed successfully.
Created Session ID = 8
Created Process ID in session = 256
```

Figure 9-3. *Output from PROG9A.C.*

Processes

In addition to their logical devices, sessions also contain one or more processes. Processes can be thought of as programs running in a session, like CMD.EXE for the OS/2 2.1 command shell executing in an OS/2 window or EPM.EXE for the Enhanced Editor program. Each process has its own memory, file handles, interprocess communication (IPC) objects (queues, semaphores, and so on), and threads. A process's memory consists of a virtual address space that is shielded from the memory of other processes as well as the system's memory.

It should be noted that sessions afford a higher degree of protection than processes. Sessions prevent applications, like DOS programs, from accessing all of the machine's resources, including memory, directly. These logical devices are shared among all the processes in the session. Note again that the terms *program* and *process* are used interchangeably in this chapter. A process is a program if the program only has one process; however, OS/2 2.1 gives you the flexibility to utilize multiple processes in your application program. The processes need not be related nor even reside in the same session.

You can create a process in one of three ways:

❑ By starting a session and creating a process by executing a program in that session.

❑ By double-clicking on an application icon in a program folder, which also starts a session before running the program.

❑ By calling the DosExecPgm API from inside a program. The DosExecPgm API creates another process in the same session. The next example demonstrates a use of this API.

In the next code example, the first program (PROG9B.C) uses the DosExecPgm API to create another program synchronously (PROG9BA.C). PROG9B.C is shown in Figure 9-2; PROG9BA.C is shown in Figure 9-3.

Since the two processes have a synchronous relationship, the first program will wait for its created program to complete before it resumes processing. This spawned program is known as a child process.

Processes are related through a process hierarchy tree with parent and child processes. A process can have only one parent, but it can have more than one child process. Use the DosGetInfoBlocks API to query a process's attributes, including its parent's process ID.

The output shows program PROG9BA.C executing in a child process, printing out "A"s until it ends. This is shown in Figure 9-4. Its parent program, PROG9B.C, takes over afterwards, printing "B"s before it prints out the return code structure from the DosExecPgm API. It also prints information from the Thread Information Block (TIB) and Process Information Block (PIB) structures. Both of the structures were retrieved by calling the DosGetInfoBlocks API.

The DosExecPgm API is detailed below. A process can create other processes with this API. These processes share the same resources, such as files, handles, and pipes, of their parent process.

The DosExecPgm API creates the process by loading the program into its virtual address space, resolving any dynamic links, calling the main entry API, and starting it with a single thread. The process's virtual address space is distinct from its parent process.

DosExecPgm

```
DosExecPgm(
            PCHAR ObjNameBuf,
            LONG ObjNameBufL,
            ULONG ExecFlags,
            PSZ ArgPointer,
            PSZ EnvPointer,
            RESULTCODES &ReturnCodes,
            PSZ pgmPointer);
```

Parameters:

❑ ObjNameBuf (PCHAR) and ObjNameBufL (LONG) input—These are the same parameters as those in the DosStartSession STARTDATA structure. (The DosStartSession API executes the DosExecPgm API in a separate session with these values.) If an error occurs, you can examine the contents of this parameter for the error text. ObjNameBufL specifies its length.

- ExecFlags (ULONG) input—Indicates how the program should run. These are possible values:

Code	Name	Description
0	EXEC_SYNC	Synchronous to the parent.
1	EXEC_ASYNC	Asynchronous to the parent.
2	EXEC_ASYNCRESULT	Asynchronous to the parent and after the process completes, its result codes are saved.
3	EXEC_TRACE	Like EXEC_ASYNCRESULT, but with trace features.
4	EXEC_BACKGROUND	Asynchronous and detached from the parent.
5	EXEC_LOAD	Program loaded, but not immediately executed.
6	EXEC_ASYNCRESULTDB	Like EXEC_ASYNCRESULT, but with trace features for the process and its its dependents. If EXEC_ASYNCRESULT, EXEC_TRACE, or EXEC_ASYNCRESULTDB is specified, you can use the DosWaitChild API to suspend your parent process until its specified child process ends.

- ArgPointer (PSZ) input—Argument string for the program of the form:

```
"programFilename\0arg1 arg2...argN\0\0"
```

- EnvPointer (PSZ) input—Environment string for the program of the form:

```
"environmentVar1\0environmentVar2\0....
    environmentVarN\0\0"
```

Note how the null terminators "\0" are used in both the Arg pointer and Env pointer parameters. You will have to construct your values in this fashion. In the output for this example, only the first environment variable string is shown, although more variables exist. (The printf function will print characters until the first NULL.) If this parameter is set to zero when the DosExecPgm API is called, the created process inherits its parent's environment.

- ReturnCodes (PRESULTCODES) output—Contains the RESULTCODES structure:

```
{ ULONG termcodepid
  ULONG resultcode
} RESULTCODES
```

  If the child is asynchronous, the termcodepid contains the Process ID (PID) of the child process. If the child is synchronous, termcodepid will contain termination code of the ending process. When the process is running synchronously, you don't need the PID of the child process since your parent process is waiting for the child to end. Possible values are:

```
0  TC_EXIT
1  TC_HARDERROR
2  TC_TRAP
3  TC_KILLPROCESS
4  TC_EXCEPTION
```

  The resultcode contains the return code of your synchronous child process. You can provide this value by using the DosExit API in your child process.

- PgmPointer (PSZ) input—Program file name to be executed.

The attributes of the current process and current thread can be identified by the DosGetInfoBlocks API. Remember that you will have at least one thread per process.

DosGetInfoBlocks

```
DosGetInfoBlocks(
                PTIB ptib,
                PPIB ppib);
```

- ptib (PTIB) output—Pointer to a Thread Information Block (TIB) for the thread that called the DosGetInfoBlocks API.

- ppib (PPIB) output—Pointer to a Process Information Block (PIB) for the process that called the DosGetInfoBlocks API.

Many of the variables from the TIB and PIB structures are shown in the example program. Note that the Process ID (PID) uniquely identifies the process to the OS/2 system and the combination of PID and Thread ID (TID) uniquely identifies the thread to the system.

PROG9BA.C also uses the DosExit API to end its processing and supply a return code.

DosExit

```
DosExit(
        ULONG ActionCode,
        ULONG ResultCode);
```

❑ ActionCode (ULONG) input—The action the API should take, ending either the process or the thread. Possible values are:

```
0  EXIT_THREAD
1  EXIT_PROCESS
```

By specifying EXIT_PROCESS, you will end the entire process even if there are more threads running.

❑ ResultCode (ULONG) input—Return code to give to the waiting process, if any.

```
/*  prog9ba.c.  Crreating a synchronous child process                */

#define INCL_DOSPROCESS          /* Include process and thread APIs */
#include <os2.h>
#include <stdio.h>               /* for the printf functions        */

VOID main (void)
{
   CHAR        ProgramLoadError[100];
   PSZ         ProgramArgs=NULL;
   PSZ         ProgramEnvs;
   CHAR        ProgramName[20];
   RESULTCODES ReturnCodes;
   APIRET      rc;
   PTIB        pptib;
   PPIB        pppib;
   INT i;

   strcpy(ProgramName,"PROG9BA.EXE");        /* Program to create     */

   /* Start Program */
   rc = DosExecPgm(ProgramLoadError,         /* Load Error Msg Buffer */
           sizeof(ProgramLoadError),
                                 /* Size of Load Error Msg Buffer */
            EXEC_SYNC,           /* Execute synchronously         */
            ProgramArgs,         /* Program Argument String       */
            ProgramEnvs,         /* Program Environment String    */
            &ReturnCodes,        /* Execution Termination codes   */
            ProgramName);        /* Program File Name             */

   /* if DosExecPgm executed successfully, print return code  info. */
   if (rc == 0) {
      printf("DosExecPgm called successfully.\n");
      printf("Termination Code = %d\n",
         ReturnCodes.codeTerminate);
      printf("Return Code = %d\n",ReturnCodes.codeResult);
   } else {
      printf("DosExecPgm ended with error code: %d\n", rc);
      return;
   }

   /* print characters                                            */
   for (i=1; i<100;i++) {
      printf("B");
   } /* endfor                                                    */
   printf("\n");

   /* Print the TIB and PIB blocks for thread 1 of this process.    */
   rc = DosGetInfoBlocks(&pptib,&pppib);
   if (rc == 0) {
      printf("\nDosGetInfoBlocks called successfully.\n");
      print_tib_pib_info(pptib,pppib);
   } else {
      printf("\nDosGetInfoBlocks ended with error code:
         %d\n", rc);
      return;
   }
   return;
}

/* print_tib_pib_info - Prints a subset of the information in the   */
/* TIB and PIB structures                                           */

int print_tib_pib_info (PTIB pptib, PPIB pppib)
{
   printf("\n*** Thread Information Block (TIB)
      Information ***\n");
   printf("TIB version number = %d\n",pptib->tib_version);
   printf("Thread ordinal number = %d\n",pptib->tib_ordinal);
   printf("\n*** Thread Information Block 2 (TIB2)Information ***\n");
   printf("Thread ID (TID) = %d\n", pptib->tib_ptib2->tib2_ultid);
   printf("Thread priority = %d\n", pptib->tib_ptib2->tib2_ulpri);
   printf("TIB2 version number = %d\n",ptib->tib_ptib2->tib2_version);
   printf("\n*** Process Information Block (PIB) Information ***\n");
   printf("Process ID (PID) = %d\n",pppib->pib_ulpid);
   printf("Parent Process ID (PPID) = %d\n",pppib->pib_ulppid);
   printf("Module Handle of Process = %d\n",pppib->pib_hmte);
   printf("Command Line String = %s\n",pppib->pib_pchcmd);
   printf("Environment String = %s\n",pppib->pib_pchenv);
   printf("Process Status Bits = %d\n",pppib->pib_flstatus);
   printf("Process Type = %d\n",pppib->pib_ultype);
}
```

Figure 9-4. *Creating a synchronous child process (PROG9B.C).*

PROG9BA.C in Figure 9-5 shows the execution of a child process.

```
/*  prog9ba.c.   Program spawned by progb.c.                       */

#define INCL_DOSPROCESS          /* Include process and thread APIs */
#include <os2.h>

main (void)
{
   int i;

   printf("I'm alive!\n");
   for (i=1;i<100;i++ ) {
      printf("A");
   }

   DosExit(EXIT_PROCESS,0);           /* Exit process with result 0 */
}
```

Figure 9-5. *Program spawned by PROG9BA.C.*

```
I'm alive!
AAAAAA....AAAAAAAABBBBBBB......BBBBB
DosExecPgm called successfully.
Termination Code = 0
Return Code = 0

DosGetInfoBlocks called successfully.

*** Thread Information Block (TIB) Information ***
TIB version number = 20
Thread ordinal number = 116

*** Thread Information Block 2 (TIB2) Information ***
Thread ID (TID) = 1
Thread priority = 512
TIB2 version number = 20

*** Process Information Block (PIB) Information ***
Process ID (PID) = 202
Parent Process ID (PPID) = 15
Module Handle of Process = 3783
Command Line String = PROG9B
Environment String = USER_INI=D:\OS2\OS2.INI
```

```
Process Status Bits = 16
Process Type = 2
```

Figure 9-6. *Output for PROG9B.C and PROG9BA.C.*

The next example, PROG9C.C in Figure 9-7, differs slightly from the last code example. PROG9C.C creates PROG9CA.C asynchronously so that both program PROG9C.C and program PROG9CA.C execute concurrently. The sample output for these programs is shown in Figure 9-8. Both programs run together for a while before program PROG9C.C waits for PROG9CA.C to terminate. Since it has only one thread, program PROG9C.C suspends itself after calling the DosWaitChild API with the PID of PROG9CA.C.

A process can wait for a process and its children or just for the process itself. Essentially, PROG9C.C could mimic the synchronous execution of the last example if it called the DosWaitChild API immediately after the DosExecPgm API. The type of returned information is similar to that in the previous example.

The DosWaitChild API suspends the current thread until the asynchronous process that it is waiting on completes.

DosWaitChild

```
          DosWaitChild(
                       ULONG ActionCode,
                       ULONG WaitOption,
                       PRESULTCODE ReturnCodes,
                       PPID PIDRetProcessID,
                       PID ProcessID);
```

Parameters:

❑ ActionCode (ULONG) input—Specifies the process scope on which to wait.

```
0  DCWA_PROCESS (process only)
1  DCWA_PROCESSTREE (process and its children)
```

❑ WaitOption (ULONG) input—Indicates whether the process should wait on the referenced child process to end.

```
0  DCWW_WAIT
1  DCWW_NOWAIT
```

- ReturnCodes (PRESULTCODE) output—Contains the RESULTCODES structure.

  ```
  { ULONG codeTerminate
    ULONG codeResult
  } RESULTCODES
  ```

  The codeTerminate variable indicates how the child ended:

  ```
  0  TC_EXIT
  1  TC_HARDERROR
  2  TC_TRAP
  3  TC_KILLPROCESS
  4  TC_EXCEPTION
  ```

  The codeResult variable contains the return code of the child process, as specified by its DosExit API. The default is -1.

- PIDRetProcessID (PPID) output—PID of the ending process.

- ProcessID (PID) input—PID of the process that the current process is waiting to end. If this value is 0, the process will wait for all the child processes of the current process to end. (Thus you will need to consult the pidRetProcessID parameter.) If the value is not 0, wait for the process identified by this PID.

```
/*  prog9c.c.  Creating an asynchronous child process (prog9ca.c).  */

#define INCL_DOSPROCESS          /* Include process and thread APIs */
#include <os2.h>
#include <stdio.h>               /* for the printf functions        */

VOID main (void)
{
   CHAR        ProgramLoadError[100];
   CHAR        ProgramArgs[100];
   PSZ         ProgramEnvs=NULL;
   CHAR        ProgramName[20];
   RESULTCODES ReturnCodes, ReturnCodes2;
   APIRET      rc, rc2;
   PTIB        pptib;
   PPIB        pppib;
   ULONG       Pid;
   INT i;

   strcpy(ProgramName,"PROG9CA.EXE");          /* Program to create */
   memset(ProgramArgs,'\0',sizeof(ProgramArgs));
   strcpy(ProgramArgs,ProgramName);
```

```
strcat(ProgramArgs,"  A");
ProgramArgs[11] = '\0';

/* Start Program */
rc = DosExecPgm(ProgramLoadError,  /* Load Error Msg Buffer      */
       sizeof(ProgramLoadError),/* Size of Load Error Msg Buffer */
       EXEC_ASYNCRESULT,          /* Execute asynchronously      */
       ProgramArgs,               /* Program Argument String     */
       ProgramEnvs,               /* Program Environment String  */
       &ReturnCodes,              /* Execution Termination codes */
       ProgramName);              /* Program File Name           */

/* if DosExecPgm executed successfully, print return code  info. */
if (rc == 0) {
   printf("DosExecPgm called successfully.\n");
   printf("Termination Code = %d\n",
      ReturnCodes.codeTerminate);
   printf("Return Code = %d\n",ReturnCodes.codeResult);
} else {
printf("DosExecPgm ended with error code = %d\n", rc);
return;
}

/* print characters                                              */
for (i=1; i<1000;i++) {
   printf("B");
}                                        /* endfor               */
/* Wait for child to finish and print out the results            */
printf("\nWaiting for child process to finish.\n");
rc2 = DosWaitChild(DCWA_PROCESS, /* Wait on the indicated process*/
   DCWW_WAIT,                    /* Wait for child to end        */
   &ReturnCodes2,                /* Execution Termination codes  */
   &Pid,                         /* Returned Ending Process ID   */
   ReturnCodes.codeTerminate) ;  /* ID of process to wait on     */

if (rc2 == 0) {
   printf("\nDosWaitChild called successfully.\n");
   printf("Termination Code = %d\n",
      ReturnCodes2.codeTerminate);
   printf("Return Code = %d\n",ReturnCodes2.codeResult);
} else {
   printf("\nDosWaitChild error: return code = %ld",rc2);
   return;
}

/* Print the TIB and PIB blocks for thread 1 of this process.    */
rc = DosGetInfoBlocks(&pptib,&pppib);
if (rc == 0) {
   printf("\nDosGetInfoBlocks called successfully.\n");
   print_tib_pib_info(pptib,pppib);
} else {
   printf("DosExecPgm ended with error code: %d\n", rc);
   return;
}
```

```
   return;
}

/*** print_tib_pib_info - Prints a subset of the information in the
TIB and PIB structures ***/

int print_tib_pib_info (PTIB pptib, PPIB pppib)
{
   printf("\n*** Thread Information Block (TIB) Information ***\n");
   printf("TIB version number = %d\n",pptib->tib_version);
   printf("Thread ordinal number = %d\n",pptib->tib_ordinal);
   printf("\n*** Thread Information Block 2 (TIB2) Information
      ***\n");
   printf("Thread ID (TID) = %d\n",pptib->tib_ptib2->tib2_ultid);
   printf("Thread priority = %d\n",pptib->tib_ptib2->tib2_ulpri);
   printf("TIB2 version number = %d\n",
      pptib->tib_ptib2->tib2_version);

   printf("\n*** Process Information Block (PIB) Information ***\n");
   printf("Process ID (PID) = %d\n",pppib->pib_ulpid);
   printf("Parent Process ID (PPID) = %d\n",pppib->pib_ulppid);
   printf("Module Handle of Process = %d\n",pppib->pib_hmte);
   printf("Command Line String = %s\n",pppib->pib_pchcmd);
   printf("Environment String = %s\n",pppib->pib_pchenv);
   printf("Process Status Bits = %d\n",pppib->pib_flstatus);
   printf("Process Type = %d\n",pppib->pib_ultype);
}
```

Figure 9-7. *Creating an asyhchronous child process (PROG9C.C).*

```
/*  prog9ca.c.   program spawned by prog9c.c                        */

#define INCL_DOSPROCESS          /* Include process and thread APIs */
#include <os2.h>

main (USHORT argc, PCHAR argv[])
{
   INT i;

   for (i=1;i<1000;i++) {
      printf(argv[1]);
   } /* endfor */

   DosExit(EXIT_PROCESS,0);           /* Exit process with result 0 */
}
```

Figure 9-8. *Program spawned by PROG9.C (CPROG9CA.C).*

```
BBBBBBBBBBBBBBB..........BBBBBBB
Waiting for child process to finish.
AAAAAAAAAAAAAA...........AAAAAAA
```

Figure 9-9. *Output for PROG9C.C and PROG9CA.C.*

What happens if you want to terminate a running process? The next example, PROG9D.C, shows how you can terminate a process before it naturally completes. (The process could have been designed to run in a loop too until it was terminated). PROG9D.C in Figure 9-10 creates another process, PROG9DA.C in Figure 9-11, asynchronously using the same technique as in the previous examples.

Both programs run for a while before program PROG9D.C with malicious and willful intent kills program PROG9DA.C with the DosKillProcess API. By using the DosKillProcess API, you can terminate a process itself or a process and its children. In this example, the process and its lone thread are deep-sixed. Finally, with no remorse, PROG9D.C displays its TIB and PIB information. The output for PROG9D.C is shown in Figure 9-12.

The DosKillProcess API terminates a process and supplies the return code of the killed process, if any.

DosKillProcess

```
          DosKillProcess(
                         ULONG ActionCode,
                         PID ProcessID);
```

Parameters:

- ❑ ActionCode (ULONG) input—Specify the scope of the kill.

```
0  DKP_PROCESSTREE (the process and its children)
1  DKP_PROCESS (the process only)
```

- ❑ ProcessID (PID) input—PID of the process to be killed. The ActionCode will indicate whether just the process or the process and its descendents are killed.

Figure 9-10 shows an asynchronous example.

```
/*  prog9d.c.   Killing an asychronous process.                    */

#define INCL_DOSPROCESS          /* Include process and thread APIs */
#include <os2.h>
#include <stdio.h>               /* for the printf functions        */

VOID main (void)
{
   CHAR        ProgramLoadError[100];
   CHAR        ProgramArgs[100];
   PSZ         ProgramEnvs=NULL;
   CHAR        ProgramName[20];
   RESULTCODES ReturnCodes, ReturnCodes2;
   ULONG       ActionCode;
   APIRET      rc, rc2;
   PTIB        pptib;
   PPIB        pppib;
   ULONG       Pid;
   INT i;                                               /* counter */

   strcpy(ProgramName,"PROG9DA.EXE");          /* Program to spawn */
   memset(ProgramArgs,'\0',sizeof(ProgramArgs));
   strcpy(ProgramArgs,ProgramName);
   strcat(ProgramArgs,"  A");
   ProgramArgs[11] = '\0';

   /* Start Program */
   rc = DosExecPgm(ProgramLoadError,        /*Load Error Msg Buffer */
        sizeof(ProgramLoadError), /* Size of Load Error Msg Buffer */
        EXEC_ASYNCRESULT,              /* Execute asynchronously */
        ProgramArgs,                  /* Program Argument String */
        ProgramEnvs,               /* Program Environment String */
        &ReturnCodes,             /* Execution Termination codes */
        ProgramName);                       /* Program File Name */

   /* if DosExecPgm executed successfully, print return code info.  */
   if (rc == 0) {
      printf("DosExecPgm called successfully.\n");
      printf("Termination Code = %d\n",
         ReturnCodes.codeTerminate);
      printf("Return Code = %d\n",ReturnCodes.codeResult);
   } else {
      printf("DosExecPgm ended with error code: %d\n", rc);
      return;
   }

   /* print characters                                              */
   for (i=1;i<4000;i++) {
      printf("B");
   }                              /* endfor                          */
   printf("\n");

   ActionCode = 1;                             /* Kill the process */
   Pid = ReturnCodes.codeTerminate;
   rc2 = DosKillProcess(ActionCode, Pid);
   if (rc2 == 0) {
      printf("DosKillProcess called successfully.\n");
   } else {
      printf("DosKillProcess error: return code = %ld",rc2);
      return;
   }

   /* Print the TIB and PIB blocks for thread 1 of this process.    */
   rc = DosGetInfoBlocks(&pptib,&pppib);
   if (rc == 0) {
      printf("\nDosGetInfoBlocks called successfully.\n");
      print_tib_pib_info(pptib,pppib);
   } else {
      printf("\nDosGetInfoBlocks ended with error code:
         %d\n", rc);
      return;
   }
   return;
}

/* print_tib_pib_info - Prints a subset of the information in the   */
/* TIB and PIB structures                                            */

int print_tib_pib_info (PTIB pptib, PPIB pppib)
{
   printf("\n*** Thread Information Block (TIB) Information ***\n");
   printf("TIB version number = %d\n",pptib->tib_version);
   printf("Thread ordinal number = %d\n",pptib->tib_ordinal);

   printf("\n*** Thread Information Block 2 (TIB2) Information
      ***\n");
   printf("Thread ID (TID) = %d\n",
      pptib->tib_ptib2->tib2_ultid);
   printf("Thread priority = %d\n",
      pptib->tib_ptib2->tib2_ulpri);
   printf("TIB2 version number = %d\n",
      pptib->tib_ptib2->tib2_version);

   printf("\n*** Process Information Block (PIB) Information
      ***\n");
   printf("Process ID (PID) = %d\n",pppib->pib_ulpid);
   printf("Parent Process ID (PPID) = %d\n",pppib->pib_ulppid);
   printf("Module Handle of Process = %d\n",pppib->pib_hmte);
   printf("Command Line String = %s\n",pppib->pib_pchcmd);
   printf("Environment String = %s\n",pppib->pib_pchenv);
   printf("Process Status Bits = %d\n",pppib->pib_flstatus);
   printf("Process Type = %d\n",pppib->pib_ultype);
}
```

Figure 9-10. *Killing an asynchronous process (PROG9D.C).*

Figure 9-11 shows an asynchronous example.

```
/* prog9da.c.  Program killed by prog9d.c.                            */

#define INCL_DOSPROCESS          /* Include process and thread APIs */
#include <os2.h>

main (USHORT argc, PCHAR argv[])
{
   int i;

   DosSleep(3);
   /* print character passed in argument string                        */
   for (i=1;i<10000;i++) {
      do
         printf(argv[1]);
      if(TRUE)
   }

}
```

Figure 9-11. *Program killed by PROG9D.C (PROG9DA.C).*

```
AAAAAAAAAAAA......AAAAAAAAAAAAA
AAAA.....AAAAAABBBBBBBBB....BBBBBBBB
BBBBBBBBBBB.........BBBBBBBBBBB
DosKillProcess called successfully.
DosExecPgm called successfully.
Termination Code = 70
Return Code = 0

DosGetInfoBlocks called successfully.
*** Thread Information Block (TIB) Information ***
TIB version number = 20
Thread ordinal number = 113
*** Thread Information Block 2 (TIB2) Information ***
Thread ID (TID) = 1
Thread priority = 512
TIB2 version number = 20
*** Process Information Block (PIB) Information ***
Process ID (PID) = 69
Parent Process ID (PPID) = 15
Module Handle of Process = 3526
Command Line String = cZZZ
Environment String = USER_INI=D:\OS2\OS2.INI
Process Status Bits = 16
Process Type = 2
```

Figure 9-12. *Output for PROG9D.C and PROG9DA.C.*

THREADS

Threads are the most basic unit of execution in OS/2 2.1. A CPU essentially executes threads, not processes or sessions. A process will always have at least one thread. You can create more than one thread to help distribute the workload of your process in an equitable fashion. The use of multiple threads becomes particularly important with the performance of PM programs. Most PM programs are particularly user-interface intensive. You can dedicate one thread to service user interface interaction and target the others to perform calculations, I/O, or other tasks. When it is called, the DosCreateThread API generates a thread for the process to use. The process for creating and maintaining threads is less expensive than that for creating additional processes or sessions.

As mentioned earlier, all the processes in a session have the ability to share the devices held by their parent session. The threads of a process are also able to share the virtual address space, I/O handles, and IPC objects that belong to their parent process. This also helps to reduce the cost of creating threads and gives OS/2 an advantage over a single-process, single-threaded operating system.

Threads also inherit their priority level from their parent process. OS/2 2.1 can dynamically change their levels at a later time. It's also easier to share information between threads than between processes. To share information between processes, use IPC objects like pipes, queues, and shared memory.

Each thread owns a set of registers, a stack, and an execution priority. OS/2 2.1 supports up to 4,096 threads. However, this number is actually limited by the THREADS parameter in your CONFIG.SYS file. Since OS/2 2.1 will also consume a number of threads for its own use, the actual number will be smaller than that specified in CONFIG.SYS. The THREADS parameter defaults to 64. It it can range from 32 to 4,096.

A process must have at least one thread—call it thread 1. Many actions are entrusted to this first thread. Should this thread die, naturally or abnormally, the entire process will terminate. It has the added responsibility of receiving the exceptions (or signals) sent to the process, as well as registering the exit list for the entire process. Asynchronous exceptions can be actions such as the user pressing control-break. If specified, the exit list gives the process a routine to perform before it terminates.

Priorities

Threads are given a CPU timeslice in which to run, but before any thread can execute, the OS/2 2.1 scheduler must choose it. This selection is made based on the priority class and level of a thread. The highest goes first, naturally. A thread priority can vary not only by a specific API calls on the thread's parent process, but by indirect dynamic changes as well.

Within each group of threads at a priority level, the scheduler dispenses CPU timeslices in a round-robin fashion. For example, if three threads were at priority level 5, the first would execute, then the second, next the third, then back to the first, and so forth in sequential fashion. This scheme ensures that all threads at the same priority have an equal chance to execute. This example also assumes that interrupts or additional threads don't enter the fray.

Before reviewing the classes and levels of priority, first examine the states at which a thread can reside. These classes are *running*, *read to run*, and *blocked*, as illustrated in Figure 9-13. Only one thread can run at a time with a single processor, which is what OS/2 2.1 currently supports. Most other threads exist in the ready-to-run state, biding their time until the scheduler grants a timeslice to them. Threads that are not running, or that are not ready-to-run, are blocked. These blocked threads are waiting for an event to occur before they can graduate to the ready-to-run state. Threads can be blocked when they are waiting for I/O to complete, for example.

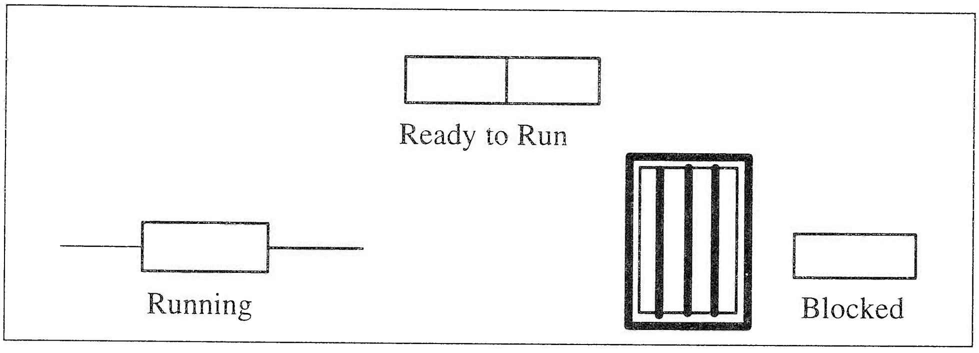


Figure 9-13. *Thread states.*

When the scheduler switches between a running and a ready-to-run thread, it saves the outgoing thread's environment, such as its registers, and restores the incoming thread's environment. The scheduler also grants the thread a timeslice equal to or less than the TIMESLICE value specified in the CONFIG.SYS file. The default timeslice is 32 milliseconds. Timeslices can range from 32 to 65,536 milliseconds. The thread usually gets the full timeslice unless an interrupt muscles its way in or unless a higher priority thread preempts the current thread.

Priorities are first divided into the following classes: time critical, server, regular, and idle. Each class is further divided into priority levels ranging from 1 to 31.

- The *time critical* class tops the hierarchy. Threads and processes assigned to this class need the responsiveness and immediacy this class affords. A communications transfer would be one such timing-critical process.

- The *server* (or fixed-high) class comes next. Processes and threads in this class are usually programs that run on a server and that should take precedence over other tasks a user at the server workstation may be running.

- Most user programs, and most programs in general will run in the next class—the *regular* class.

- The lowest class of all, the *idle* class, usually encompasses daemon programs and other non-critical processes. They typically sleep and periodically come to life when the processor is free of other, more important, processes. Some electronic mail programs use a daemon process that wakes during low CPU usage to check for mail addressed to the user from the mainframe or across a LAN.

Threads inherit their priority levels from the priority given to their parent process. Developers can also change the priority of a process or thread with the DosSetPriority API. If left unchanged, a thread will run at the given priority unless the system dynamically changes it. These dynamic changes are called priority boosts since they have a positive effect on the thread's priority, even if it is only temporary.

Even if you use the DosSetPriority API on a thread, OS/2 2.1 will gradually change the thread's priority over time. For example, if it's too low and doesn't run for a while, OS/2 2.1 will boost its priority so it gets a chance to run in the CPU. Other types of boosts include *foreground*, *I/O*, and *starvation*. A *foreground boost* applies to the session, and thus includes the process and the threads in the process that are running in the foreground. Typically, this would be the process that you currently have your attention on. Because of this, the session should be more responsive and perform better than background processes. This boost lasts as long as the process is in the foreground.

An *I/O boost* comes to the aid of a thread that has been in a blocked state. These threads generally have been waiting for an I/O device to finish and, therefore, they may not have had the opportunity to run for a while. With this boost, the thread is moved to a higher priority, spurring on the scheduler to assign it a timeslice sooner.

Foreground and I/O boosts help performance. *Starvation boosts* are required out of necessity. This boost prevents a thread from being locked out of the processor for an unconscionable amount of time. This boost applies to threads in the regular class that have been in the ready-to-run state for a substantial period without being given the chance to run. The MAXWAIT parameter in the CONFIG.SYS file determines how long the scheduler will wait until the thread is fed this boost. This value defaults to 3 seconds and it can range from 1 to 255. The scheduler temporarily elevates the starved thread to a higher priority, and the thread can now indulge.

OS/2 2.1 additionally employs a dynamic timeslicing algorithm which reduces the number of interrupts. This allows threads to use more of the time allotted to them. It also includes the number of threads in the ready-to-run state, as part of its scheduling algorithm.

With both threads and processes introduced, the next example will conduct a horse race between threads of different priorities in the same process. Each process in OS/2 starts with one thread, but additional ones can be created with the DosCreateThread API. The threads in a process are siblings to each other. Each thread's priority defaults to the process's priority, but the DosSetPriority API can be used to change it. The random number function called in PROG9E.C gives a boost in priority between 1 and 31. PROG9E.C is shown in Figure 9-14. (Priorities can be set across separate processes, or they can be made to affect all the threads in a process.)

Note that negative priority deltas are passed to the DosSetPriority API even though it does not lower the thread's priority below 1. Having some threads at a higher priority level and some at the same level shows how threads react with one another. In the output for PROG9E.C, each thread prints out its unique letter to show how the different priorities work. The output is shown in Figure 9-15.

Though you may have a favorite thread, the program is— brace yourself—rigged, but only sightly, depending on your point of view and whether or not your thread won. A thread created before another gives it an advantage over later threads. With all threads started in suspended state, the DosResumeThread API kicks them off, but in sequential order.

The threads could start closer together if they waited on a semaphore or on a global variable value to change. Of course, the starting order advantage could also be rendered insignificant by the priority assigned to the thread. There is a great gulf between priority levels 1 and 31. Also note that these threads (albeit the entire process) execute in the regular priority class, the default for most processes.

This program could be enhanced by displaying the TIB information for each thread using the DosGetInfoBlocks API and displaying the values within a critical segment using the DosEnterCritSeg and DosExitCritSeg APIs. This insures that another thread in the process can't seize the processor until the thread exits the critical segment. These segments should be kept very short since they have the potential to impact the program's performance if used extensively. These three APIs can be used to show each thread's TIB upon its completion.

Further enhancements could introduce an additional scenario: Misfortune, perhaps the rise of video gambling and lotteries, has caused the demise of the "racetrack" used in PROG9E.C. Yet over time, interest in the thread racetrack has been rekindled and new owners have opened the racetrack for business once again. Sadly, the new management failed to effect repairs on the field and the racetrack has become quite treacherous to the threads that run upon it. A thread can find itself quickly mired in a mud-pit before eventually breaking free, or a thread can meet a tragic end by falling into a bottomless pit. Unlike the previous racetrack, there may not be a winner.

This enhancement would use the same API calls as in PROG9E.C, but now it must represent a thread getting stuck in a mud-pit with the DosSuspendThread API and freeing itself with the DosResumeThread API. The DosKillThread would simulate the bottomless pits into which a thread could plummet. The appearance of these pits would, of course, be random, and they may not claim any thread, but that wouldn't be very interesting (or amusing to the perspiring and slightly inebriated crowd).

Alternatively, the _beginthread and _endthread functions in the IBM C Set ++ compiler could be used in place of the DosCreateThread and DosKillThread APIs. You should use the DosCreateThread API since it offers better performance overall.

The DosCreateThread API creates an asynchronous thread that will execute in the current process.

DosCreateThread API

```
          DosCreateThread(
                          PTID ThreadID,
                          PFNTHREAD ThreadAddr,
                          ULONG ThreadArg,
                          ULONG ThreadFlags,
                          ULONG StackSize);
```

Parameters:

- ❑ ThreadID (PTID) output—TID of the created thread.
- ❑ ThreadAddr (PFNTHREAD) input—Function that will be executed by the thread.
- ❑ ThreadArg (ULONG) input—Arguments to pass the thread, such as those that will be used by the thread function.
- ❑ ThreadFlags (ULONG) input—If bit 0 is 0, the thread starts when called. If bit 0 is 1, the thread starts in a suspended state. The DosResumeThread API must be called to unsuspend the thread. If bit 1 is 0, the default stack creation method will be used. If bit 1 is set to 1, all pages in the stack are committed.
- ❑ StackSize (ULONG) input—Size of the stack the thread uses, measured in bytes.

The DosResumeThread API allows a suspended thread to resume execution.

DosResumeThread

```
          DosResumeThread(
                          TID ThreadID);
```

Parameter:

- ❑ ThreadID (TID) input—TID of the thread to resume.

The DosSetPriority API sets the priority of a thread or process that is a child of the current process.

DosSetPriority

```
DosSetPriority(
              ULONG Scope,
              ULONG PriorityClass,
              ULONG PriorityDelta,
              ULONG ID);
```

Parameters:

- ❑ Scope (ULONG) input—Scope of objects that the priority will change.

```
0  PRTYS_PROCESS (all the threads of the process)
1  PRTYS_PROCESSTREE (all the threads of the current
      process and its descendents)
2  PRTYS_THREAD (the specified thread only)
```

- ❑ PriorityClass (ULONG) input—Priority class that should be set.

```
0  PRTYC_NOCHANGE
1  PRTYC_IDLETIME
2  PRTYC_REGULAR
3  PRTYC_TIMECRITICAL
4  PRTYC_FOREGROUNDSERVER
```

- ❑ PriorityDelta (ULONG) input—Parameter to change the priority of the object from -31 to +31.

- ❑ ID (ULONG) input—ID of the object (PID for process or TID for thread) to change. If set to 0, the ID defaults to the current process.

```
/* prog9e.c.  Creating threads.                              */

#define INCL_DOSDATETIME                      /* Date and time values */
#define INCL_DOSPROCESS              /* Include process and thread APIs */
#include <os2.h>
#include <stdio.h>                        /* for the printf functions */
```

```
#include <ctype.h>                          /* for the toascii function */
#include <stdlib.h>                         /* for the rand functions   */

#define NUMTHREADS 4

VOID thread_func (ULONG);              /* thread function declaration */

VOID main (void)
{
   APIRET rc;
   INT   i;
   INT   rnum;                                  /* random number       */
   ULONG seed;                                  /* random number seed */
   ULONG threadArgs;
   ULONG threadFlags;
   ULONG threadStackSize;
   TID   thread_id[NUMTHREADS];        /* array of createdthread IDs */
   DATETIME  DateTime;
   ULONG   priScope;                  /* scope of priority change    */
   ULONG   priClass;                  /* class to set priority by    */
   LONG    priDelta;                  /* delta to change priority by */

   threadArgs = 0L;                   /* No arguments                 */
   threadFlags = 1L;        /* Start the thread in a suspended state */
   threadStackSize = 4096;         /* Make the stack size a 4K block */

   /* first seed the random number generator                        */
   rc = DosGetDateTime(&DateTime);            /* Date/Time structure */
   seed = DateTime.hundredths;
   srand(seed);
   printf("Random seed = %d.\n",(seed*-1));

   for (i=0;i<NUMTHREADS;i++) {
      rc = DosCreateThread(&thread_id[i],           /* TID returned */
         (PFNTHREAD)thread_func,       /*Address of thread function */
         threadArgs,              /* Arguments to pass to the thread */
         threadFlags,             /* Flags to start the thread       */
                                  /* running or suspended            */
         threadStackSize);

      if (rc != 0) {
         printf("Error calling DosCreateThread with return
           code = %ld", rc);
         return;
      }

      printf("#%d : TID = %d\n", i, thread_id[i]);
   }
   /* Randomly set priorities                                       */
   for (i=0;i<NUMTHREADS;i++) {
      do {
         rnum = rand() - 31;
      } while ((rnum > 31) || (rnum < -31));
      printf("Rand = %d\n",rnum);
      priScope = PRTYS_THREAD;          /* Only change the thread's  */
                                        /* priority                  */
      priClass = PRTYC_NOCHANGE;        /* Keep thread in the same   */
                                        /* class                     */
      priDelta = rnum;          /* Modify the priority with a delta */

      rc = DosSetPriority(priScope, priClass, priDelta,
          thread_id[i]);

      if (rc != 0) {
         printf("DosSetPriority error: return code = %ld", rc);
         return;
      }
   }
   /* Start all threads that were created and suspended */
   for (i=0;i<NUMTHREADS;i++) {
      DosResumeThread(thread_id[i]);
   }
   /* Wait for the threads to complete                              */
   for (i=0;i<NUMTHREADS;i++) {
      printf("\nWaiting for thread #%d.\n",thread_id[i]);
      DosWaitThread(&thread_id[i], 0);
   }

   return;
}

VOID thread_func (ULONG Parm)
{
   int i,j;
   TID tid;
   APIRET rc;
   PTIB    pptib;                           /* Pointer to the TIB */
   PPIB    pppib;                           /* Pointer to the PIB */
   UCHAR ch;

   /* find out what thread it is and the character used             */
   rc = DosGetInfoBlocks(&pptib,&pppib);
   tid = pptib->tib_ptib2->tib2_ultid;                 /* thread id */
   ch = toascii(32+tid);           /* unique character to print out */
   printf("\nStarting TID #%d with character %c.\n",tid,ch);

   /* print out characters with a slight pause in between           */
   for (i=1;i<1000;i++) {
      DosSleep(10);                        /* 10 millisecond delay */
      printf("%c",ch);
   }
   /* Herald the end                                                */
   printf("\nTID #%d Complete\n",tid);
   DosExit(EXIT_THREAD, 0);           /* Exit thread with result 0 */
}
```

Figure 9-14. *Creating threads (PROG9E.C).*

```
Random seed = -53.
#0 : TID = 2
#1 : TID = 3
#2 : TID = 4
#3 : TID = 5
    Rand = 5
Rand = -22
Rand = -8
Rand = 27

Starting TID #2 with character ".
"""""".......................""""""" 
"""""".......................""""""" 
TID #2 Complete

Starting TID #5 with character %.
%%%%%%............%%%%%%%%%
%%%%%%............%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%
TID #5 Complete

Waiting for thread #2.

Waiting for thread #3.

Starting TID #3 with character #.
#####............##########
Starting TID #4 with character $.
$$$$$.............$$$$$$$$$$$
#######..............###########
#$$$$$$$$.........$$$$$$$$$$$
######.......###########
#######.....#######$$$$$$$$......$$$$$$$$$
$$$$.........$$$$$$$$
$#$#$#$#$#$#$#..........$#$#$#$#$#$#
.....$$$$
TID #4 Complete
###########.........####################
TID #3 Complete

Waiting for thread #4.

Waiting for thread #5.
```

Figure 9-15. Output for PROG9E.C.

API SUMMARY

Program execution control APIs, both those used in examples in this chapter and those that weren't, are listed below:

Session APIs

- ❑ DosSelectSession—Switches a child session to the foreground. The parent must be executing in the foreground when this API is called.

- ❑ DosSetSession—Sets the status of a child session, including its selectable and bonding attributes. A bonded child moves to the foreground when its parent is selected.

- ❑ DosStartSession—Starts an independent or child session in the foreground or background..

- ❑ DosStopSession—Terminates one or more child sessions.

Process and Thread APIs

- ❑ DosCreateThread—Creates a thread within a process.

- ❑ DosEnterCritSec—Prevents another thread in the process from interrupting the current one.

- ❑ DosExecPgm—Creates a process from another process.

- ❑ DosExit—Terminates a process or thread from within itself and provides a return code.

- ❑ DosExitCritSec—Allows other threads in the process to switch amongst themselves.

- ❑ DosExitList—Registers routines to run when a process ends.

- ❑ DosGetInfoBlocks—Retrieves the TIB and PIB of the current thread and process.
- ❑ DosKillProcess—Terminates a specified process.

- ❑ DosKillThread—Terminates a specified thread.

- ❑ DosResumeThread—Places a suspended thread in running state.

- ❑ DosSetPriority—Changes the priority class and level of a specified process or thread.
- ❑ DosSuspendThread—Places a running thread in a suspended state.
- ❑ DosWaitChild—Waits for a process to end.
- ❑ DosWaitThread—Waits for a thread to end.

TIPS

The benefits of constructing a program to exploit multitasking are evident when these APIs are used judiciously. Not every application needs to create processes and threads for each of its tasks. In fact, if you include multitasking APIs as part of your application, you take on additional responsibilities. Programs with multiple threads are harder to debug, and their use can introduce other problems, such as deadlocks on resources. (You can use the debug and trace options with the multitasking APIs to help track down errors.) A straightforward, sequential program doesn't need any additional multitasking other than that provided by the system.

However, on large programs, particularly user interface ones, granular multitasking will play a key part in the responsiveness of your program to the user. Whether it's a graphical or command line program, insuring that your response times are short will keep users from becoming dissatisfied with the performance of the application.

If parts of your application can be divided into separate, definable tasks that need their own address space and protection from other programs, and if they can run concurrently, then create another process. If a task can reside within the same process, and if it has a clearly defined function that can be run at the same time as other tasks (as well as run in the background), spin it off as a thread. You can create and terminate threads and processes at will, but remember that threads consume fewer resources than processes. Use processes if they will afford better protection to the task.

By their nature, applications using multiple threads are harder to debug. If a routine is used by multiple threads and an error occurs, how can you tell which thread caused the problem? You can use debuggers, such as IPMD.EXE, which comes with the IBM C Set ++ compiler, to hunt down different threads. Since IPMD requires debug compiling and linking flags to be used, eventually you have to wean your program off debug code and test your real production code. This is usually when timing and other system wide problems can occur. Use the following tips to steer clear of potential multitasking problems:

❑ Avoid setting global variables within code that multiple threads can call. Better yet, avoid using global variables altogether. Otherwise, thread X can call a procedure and set global variable A before thread Y can access the old value in variable A.

❑ Serialize access to critical portions of your code. You can easily accomplish this with event semaphores. For example, let's say a procedure posts a LAN logon pop-up when called. If multiple threads can call this procedure, use a semaphore to ensure that only one call to the procedure will execute at a time. Otherwise, you may wind up with more than one logon pop-up. This can confuse your user and could cause run-time problems if not all the logon pop-ups are closed.

❑ Be wary in using the idle class priority for threads. Depending on the load on your system, your thread may not get executed for a considerable period of time. Unless you can prove that running in the idle class helps your performance, use the regular class instead.

❑ If you have to include debug statements within your code, avoid using printf function calls. If your program traps, you can't review the statements that were printed even if you were redirecting the output to a file. Instead, use a system-wide procedure in your code to open a trace file, write the trace information to the end of the file, and close the file immediately afterwards. This way, you won't lose your printed trace statements in the event of a crash. If you print the time, process ID, and thread ID with each statement, you can follow the execution path of each of your processes and threads and can find your errors quickly. Your trace statements can log the entry and exit to each function call, errors from return codes, and other information.

For easier porting between platforms, remember that OS/2 is a multi-process environment, which allows multiple threads per process. If you move your program to another system, some of the program's multitasking code will need to be reworked. For example, certain implementations of UNIX allow only one thread per process. Nevertheless, remember that your investment in multi-threaded applications will pay performance dividends in the long run.

SUMMARY

With sessions, processes, and threads, OS/2 2.1 provides you hierarchical control over the execution of your applications and tasks within those applications. Multitasking is another benefit of moving into the 32-bit world. OS/2 will handle it for you, but it also gives you the opportunity to control multitasking at both the system and the user level.

CHAPTER 10

Semaphores

"To use the same words is not a sufficient guarantee of understanding; one must use the same words for the same genus of inward experience; ultimately, one must have one's experiences in common." —*Nietzsche*

INTRODUCTION

Once you have decided that your program design includes multiple threads, multiple processes, or both, you may have discovered that you have a need to serialize access to some program resources and to signal other threads that an event has completed.

Resources such as arrays, files, or the screen may require serialized access if more than one thread needs to use the same resource.

One use of a semaphore is to protect the resource from being referenced or changed during certain periods where the thread needs to own access to the resource temporarily. Semaphores can also be used to notify other threads of events that have occurred.

So what is a semaphore? It's analogous to a global variable that is used to determine if you are allowed to do something. The program sets the global variable when the resource is being used and unsets it when access to the resource is no longer required.

Any thread wanting access to the resource must check first to see if the global variable is set before using the resource.

Figure 10-1 on the next page is a program that illustrates this concept.

```
#define INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>

#define STACKSIZE 4096

ULONG ulScreenFlag = 0;

VOID PrintABC(PVOID arglist);
VOID PrintDEF(PVOID arglist);

INT main(VOID)
{

   _beginthread(PrintABC, NULL, STACKSIZE, NULL);
   _beginthread(PrintDEF, NULL, STACKSIZE, NULL);
   DosSleep(5000);
   return(0L);
}

VOID PrintABC(PVOID arglist)
{

   USHORT i = 0;

   while (i < 10) {
      if (ulScreenFlag == 0) {
         DosSleep(1);
         ulScreenFlag = 1;
         printf("A");
         DosSleep(1);
         printf("B");
         DosSleep(1);
         printf("C\n");
         ulScreenFlag = 0;
         ++i;
      }
   }
}

VOID PrintDEF(PVOID arglist)
{
   USHORT i = 0;

   while (i < 10) {
      if (ulScreenFlag == 0) {
         DosSleep(1);
         ulScreenFlag = 1;
         printf("D");
         DosSleep(1);
         printf("E");
         DosSleep(1);
         printf("F\n");
```

```
            ulScreenFlag = 0;
            ++i;
        }
    }
}
```

Figure 10-1. *Using a global variable to protect a resource.*

This program has two threads which are both wanting to write some data to the screen. Before each program attempts to write to the screen, it first checks to see the value of ulScreenFlag. If this value is 0 the thread assigns its value to 1 and then uses the resource and finishes by assigning ulScreenFlag back to 0 so another thread can use the resource. Each thread constantly loops until it is able to write to the screen 10 times.

The DosSleep API in the main routine is used to allow the two threads enough time to complete. The DosSleep API calls in the two threads are used to cause the thread to give up its timeslice at that moment within the thread's processing. The value 1 is passed to the DosSleep API to indicate the thread should sleep for 1 millisecond.

There are two major problems with a program like this. One is that each thread is constantly checking the ulScreenFlag variable. This polling technique takes away CPU time from all the other threads running in the system. The most severe problem is that it doesn't protect the resource.

It is possible for this program to produce a mix of "ABC" and "DEF" on a given line of the output. Something like "DAEBFC" is possible on one of the screen's lines of output. This happens because threads are given timeslices in which to run. The thread may be interrupted right after it checks for ulScreenFlag to be 0 and before it has time to set it to 1.

When this occurs, the other thread may be given enough time to pass the test, if ulScreenFlag equals 0, as well. Therefore, both threads think they own the resource and they begin writing to the screen. The DosSleep API calls within the threads are positioned within the code to show this difficulty.

To solve this problem, you would need to code the program something like that found in Figure 10-2 on the next page.

```
/* The use of hypothetical ScreenAccess functions.           */

#define INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>

#define STACKSIZE 4096

VOID PrintABC(PVOID arglist);
VOID PrintDEF(PVOID arglist);

INT main(VOID)
{

   _beginthread(PrintABC, NULL, STACKSIZE, NULL);
   _beginthread(PrintDEF, NULL, STACKSIZE, NULL);
   DosSleep(5000);
   return(0L);
}

VOID PrintABC(PVOID arglist)
{
   USHORT i = 0;

   while (i < 10) {
      WaitForScreenAccess();
      printf("A");
      printf("B");
      printf("C\n");
      ReleaseScreenAccess();
      ++i;
   }
}

VOID PrintDEF(PVOID arglist)
{
   USHORT i = 0;

   while (i < 10) {
      WaitForScreenAccess();
      printf("D");
      printf("E");
      printf("F\n");
      ReleaseScreenAccess();
      ++i;
   }
}
```

Figure 10-2. *Program illustrating the use of hypothetical ScreenAccess functions.*

The WaitForScreenAccess function should cause the thread to sleep until it is granted access to the screen. It will simultaneously wake up the thread and give it the ownership of the screen. Once the ReleaseScreenAccess function is called, the screen can be used by another thread.

The problem is, how do you write the function WaitForScreenAccess? This function needs to issue two commands at virtually the same time. Since this isn't possible without parallel processing, the operating system provides you with a tool for serializing access to resources.

Semaphores are tools that are available to solve both of these problems. With a semaphore, the thread waits until it is granted access to the semaphore, which allows your program to use the resource it protects. No polling is done. The thread is asleep until this occurs. When a semaphore becomes unowned and a thread is requesting ownership, the operating system simultaneously wakes up the thread that was waiting on the semaphore and sets that thread as the owner of the semaphore. Therefore, the thread is not interrupted in between these two steps.

In OS/2 2.1, there are the *mutual exclusion*, *event*, and *multiple wait* categories of semaphores. Each category of semaphore allows the creation of a semaphore that is named or unnamed and shared or private. Giving a name to a semaphore gives the application designer the ability to easily communicate the semaphore to another process.

With this name, another process can easily query the semaphore handle that is associated with the name and can therefore use the semaphore to access the resource. Without this name, it is still possible to access the semaphore from another process, but the other process has to know the semaphore handle which is returned from the DosCreate*Sem API.

An OS/2 named semaphore must have a name with \SEM32\ as its prefix (for example, \SEM32\mysem). A named semaphore is always a shared semaphore. A shared semaphore is one that is created for the purpose of being referenced by more than one process. The number of shared semaphores available within the system is limited to 64K and therefore should be used only if multiple process usage is required. An indivdual process can use up to 64K of private semaphores. The following sections in this chapter discuss and give example to the three categories of semaphores.

MUTUAL EXCLUSION SEMAPHORES

Mutual exclusion semaphores are used to serialize access to resources, such as tables or files, that are shared among threads. They will allow only one thread to access the resource at a time. When a thread wants access to the resource, it requests ownership of the semaphore. The system will block the thread until it is granted ownership of the semaphore. Once the thread has been given ownership of the semaphore and it is finished using the protected resource, it releases ownership of the semaphore. In OS/2 2.1, there are six APIs that are available for managing a mutex (mutual exclusion) semaphore.

- ❑ DosCreateMutexSem
- ❑ DosRequestMutexSem
- ❑ DosReleaseMutexSem
- ❑ DosCloseMutexSem
- ❑ DosOpenMutexSem
- ❑ DosQueryMutexSem

The first step in using a mutex semaphore is to create the semaphore using the DosCreateMutexSem API. At this point, your design for this semaphore (shared or private and named or unnamed) is described in the parameters passed to the API.

DosCreateMutexSem

```
DosCreateMutexSem(
                  PSZ pszName,
                  PHMTX phmtx,
                  ULONG flAttr,
                  BOOL32 fState);
```

Parameters:

- ❑ pszName (PSZ) input—The name for the semaphore. If NULL is passed, the semaphore is unnamed.

- ❏ phmtx (PHMTX) output—A pointer to a storage location where the semaphore handle is returned.

- ❏ flAttr (ULONG) input—Private or shared semaphore.

```
0   Private
1   Shared
```

- ❏ fState (BOOL32) input—Semaphore ownership state.

```
TRUE    Owned
FALSE   Unowned
```

The example in Figure 10-3 illustrates the creation of a private, unnamed mutex semaphore.

```
HMTX hmtxMySem = (HMTX)NULL;      /* Global variable where a  */
                                  /* mutext sem is stored     */

APIRET rc;                    /* Local variable where an API  */
                              /* return code is stored        */

rc = DosCreateMutexSem((PSZ)NULL,             /* Unnamed sem */
            &hmtxMySem,       /* Address for semaphore handle */
            0,                           /* Private semaphore */
            FALSE);                      /* Initially unowned */
```

Figure 10-3. *DosCreateMutexSem introduction.*

Now that the semaphore is created, the threads in the same process have the ability to request and release their use of the semaphore. The DosRequestMutexSem API is used to request ownership of the semaphore by the calling thread. Once the caller is granted ownership, the thread can access the resource.

When the use of the resource that is being protected by the semaphore is complete, the DosReleaseMutexSem API is used to return the semaphore back to being unowned.

DosRequestMutexSem

```
          DosRequestMutexSem(
                             HMTX hmtx,
                             ULONG ulTimeout);
```

Parameters:

- ❑ hmtx (HMTX) input—Semaphore handle.
- ❑ ulTimeout (ULONG) input—The number of milliseconds to wait for the semaphore to be released, or SEM_INDEFINITE_WAIT, which means to wait forever.

DosReleaseMutexSem

```
DosReleaseMutexSem(
                   HMTX hmtx);
```

Parameter:

- ❑ hmtx (HMTX) input—Semaphore handle.

The example in Figure 10-4 illustrates this use.

```
rc = DosRequestMutexSem(hmtxMySem,              /* semaphore handle */
         SEM_INDEFINITE_WAIT);                 /* Wait while in use */

/* Access the resource that is being protected                       */
     .
     .
     .

rc = DosReleaseMutexSem(hmtxMySem);             /* semaphore handle */
```

Figure 10-4. *DosRequestMutexSem and DosReleaseMutexSem.*

In the above example, the DosRequestMutexSem API is called with the SEM_INDEFINITE_WAIT parameter. With this option, the API will not return until the semaphore has been released by the thread that currently owns it.

This parameter can also be the number of milliseconds to wait before ending the attempt to access the resource at this time. If this time elapses before ownership of the semaphore can be granted, ERROR_TIMEOUT is returned.

Depending on the resource that is being protected, you may not be able to count on getting ownership of the semaphore each and every time you request it. When using

SEM_INDEFINITE_WAIT, your thread is suspended pending the release of the semaphore which might interrupt the other services that the calling thread is responsible for managing.

After the DosReleaseMutexSem API is called, the semaphore is set to unowned and any thread can then request it. If a second thread happened to be waiting for the DosRequestMutexSem to return, this would occur after the first thread issued the DosReleaseMutexSem API.

The DosRequestMutexSem API can be called multiple times by the owning thread without calling the DosReleaseMutexSem API. However, the owning thread's use of the semaphore is not released unless there is an equal number of calls to the DosReleaseMutexSem API. This is a convenience feature for developing modular, recursive, and object-based programs. The DosQueryMutexSem API is used to query the number of times the DosRequestMutexSem API has been called without a call to DosReleaseMutexSem.

When the semaphore is no longer required by any of the threads in the calling process, it should be closed. The DosCloseMutexSem API is used to close a process's use of a semaphore. Once the semaphore is closed by all processes that had referenced it, the system deletes the semaphore.

DosCloseMutexSem

```
DosCloseMutexSem(
                HMTX hmtx);
```

Parameter:

- ❑ hmtx (HMTX) Input—Semaphore handle.

A mutex semaphore is created by one process. If another process needs to use the semaphore, use the DosOpenMutexSem API instead of the DosCreateMutexSem API. After the DosOpenMutexSem API is issued, that process can issue the DosRequestMutexSem and the DosReleaseMutexSem APIs, just as the process that issued the DosCreateMutexSem API can. After the process that issued the DosOpenMutexSem no longer needs the semaphore, use the DosCloseMutexSem API to close it.

The DosQueryMutexSem API is used to request information about the semaphore. This API returns the process identifier and thread identifier of the semaphore's owner.

The API also returns the number of times that the DosRequestMutexSem API has been called on the owning thread without a matching call to the DosReleaseMutexSem on the same thread. Without an equal number of calls on the owning thread, the semaphore remains owned by that thread.

Figure 10-5 is a simple program that illustrates the use of a mutex semaphore to protect a resource. In this example the resource is the screen. The program first creates a mutex semaphore. When the semaphore is available, the program creates 10 threads that each want to access the resource.

A DosSleep call is put at the end of the program in order to give each thread enough time to complete. When all the threads are gone, the semaphore is closed.

```
/* Multiple-threaded resource management.                         */

#define INCL_DOSSEMAPHORES
#define INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>
#define STACKSIZE 4096

VOID PrintABC(PVOID arglist);
HMTX hmtxMySem = (HMTX)NULL;

INT main(VOID)
{
   USHORT i;

   DosCreateMutexSem((PSZ)NULL, &hmtxMySem, 0, FALSE);
   for ( i = 0; i < 10; ++i)
      _beginthread(PrintABC, NULL, STACKSIZE, NULL);
   DosSleep(5000);
   DosCloseMutexSem(hmtxMySem);
   return(0L);
}

VOID PrintABC(PVOID arglist)
{
   DosRequestMutexSem(hmtxMySem, SEM_INDEFINITE_WAIT);
   printf("A");
   printf("B");
   printf("C ");
   DosReleaseMutexSem(hmtxMySem);
}
```

Figure 10-5. *Multiple-threaded resource management using mutex semaphores.*

This program produces the following output to the screen:

```
ABC ABC ABC ABC ABC ABC ABC ABC ABC ABC
```

Notice that each thread had the opportunity to display *ABC*—an individual thread did not care what the other threads were doing, only whether it could have access to the resource. Also, an individual thread did not care in what order it was given access to the resource.

Consider how the program would execute without having the DosRequestMutexSem and DosReleaseMutexSem APIs. The output would be somewhat random based on the state of the machine it was running on. It is possible that you would get the following as your output:

```
AAAAAAAAAABBBBBBBBBBCCCCCCCCCC
```

Other, more likely, possibilities exist as well.

EVENT SEMAPHORES

Event semaphores are used as a signaling device to signal a thread to give up a resource or cause an event to occur. A common use of event semaphores is to have one thread wait on an event semaphore until another thread has had the opportunity to complete an assignment that was requested by the thread that is waiting on the semaphore. In OS/2 2.1 there are seven APIs used for managing an event semaphore.

- ❑ DosCreateEventSem
- ❑ DosWaitEventSem

- ❑ DosPostEventSem
- ❑ DosCloseEventSem
- ❑ DosOpenEventSem
- ❑ DosQueryEventSem
- ❑ DosResetEventSem

The first step in using an event semaphore is to create the semaphore using the DosCreateEventSem API. At this point your design for this semaphore (shared or private and named or unnamed) is described in the parameters passed to the API.

Also at this point, the design needs to consider whether the semaphore should be initially *set* or *posted*. Creating the semaphore initially set indicates the event that the semaphore represents has not occurred.

DosCreateEventSem

```
          DosCreateEventSem(
                            PSZ pszName,
                            PHEV phev,
                            ULONG flAttr,
                            BOOL32 fState);
```

Parameters:

- ❑ pszName (PSZ) input—The name for the semaphore. If NULL is passed then the semaphore is unnamed.
- ❑ phev (PHEV) output—A pointer to a storage location where the semaphore handle is returned.
- ❑ flAttr (ULONG) input—Private or shared semaphore.

```
0` Private
1  Shared
```

- ❑ fState (BOOL32) input—Semaphore state; two values are possible.

```
TRUE    Posted
FALSE   Set
```

Figure 10-6 illustrates the creation of a private, unnamed event semaphore that is initially set.

```
HEV hevMyEvent = (HEV)NULL;              /* Global variable where an  */
                                         /* event sem is stored       */

APIRET rc;                           /* Local variable where an API  */
                                     /* return code is stored        */

rc = DosCreateEventSem((PSZ)NULL,                      /* Unnamed sem */
             &hevMyEvent,            /* Address for semaphore handle */
             0,                      /* Private semaphore            */
             FALSE);                 /* Initially set                */
```

Figure 10-6. *DosCreateEventSem introduction.*

Now that the event semaphore is created and is initially set, a thread can wait on another thread to post the semaphore. To wait until an event semaphore is posted, the DosWaitEventSem API is used. One of the design decisions is to determine how long to wait. The DosWaitEventSem API allows the caller to wait indefinitely or to specify the number of milliseconds to wait. The ERROR_TIMEOUT value will be stored in the return code should the number of milliseconds of time elapse without the posting of the semaphore.

DosWaitEventSem

```
        DosWaitEventSem(
                        HEV hev,
                        ULONG ulTimeout);
```

Parameters:

- ❑ hev (HEV) input—Semaphore handle.

- ❑ ulTimeout (ULONG) input—The number of milliseconds to wait for the semaphore to be posted, or SEM_INDEFINITE_WAIT (-1L), wait forever.

After the event has occurred, the other thread can post the event semaphore, thus waking up the other thread that was waiting. The DosPostEventSem API is used to post an event semaphore.

DosPostEventSem

```
DosPostEventSem(
                    HEV hev);
```

Parameter:

- ❑ hev (HEV) input—Semaphore handle.

When the semaphore is no longer needed, it should be closed. You should use the DosCloseEventSem API to terminate a process's use of an event semaphore. If this is the last process referencing this semaphore, it will be removed from the system.

DosCloseEventSem

```
DosCloseEventSem(
                    HEV hev);
```

Parameter:

- ❑ hev (HEV) input—Semaphore handle.

Event semaphores can be accessed in other processes. If this is required, the other processes should call the DosOpenEventSem API with the event semaphore's name or handle.

After the semaphore is opened, it can be used just as the above examples have illustrated. When the semaphore is no longer required in the other process, the DosCloseEventSem API should be called.

The DosQueryEventSem API is used to query the number of times the event semaphore has been posted. The DosResetEventSem API is used to reset the number of posted times to 0. When the number of posted times is 0, the DosWaitEventSem will no longer have to wait for a DosPostEventSem to happen.

Figure 10-7 is a very concise program that shows an example usage of an event semaphore. In this example, an event semaphore is created and another thread is started. The second thread is responsible for posting the event semaphore. The first thread is responsible for waiting for the semaphore. The posting and waiting process is then repeated.

```
/*   prog10f7.  Event semaphore.                              */

#define INCL_DOSSEMAPHORES
#include <os2.h>
#include <stdlib.h>

#define STACKSIZE 4096

VOID Thread1(PVOID arglist);
HEV hevMyEvent = (HEV)NULL;

INT main(VOID)
{
   ULONG cPosts;                              /* Count of posts */

   printf("Starting program\n");
   DosCreateEventSem((PSZ)NULL, &hevMyEvent, 0, FALSE);
   _beginthread(Thread1, NULL, STACKSIZE, NULL);
   DosWaitEventSem(hevMyEvent, SEM_INDEFINITE_WAIT);
   printf("Semaphore posted\n");
   DosResetEventSem(hevMyEvent, &cPosts);
   _beginthread(Thread1, NULL, STACKSIZE, NULL);
   DosWaitEventSem(hevMyEvent, SEM_INDEFINITE_WAIT);
   printf("Semaphore posted\n");
   DosCloseEventSem(hevMyEvent);
   return(0L);
}

VOID Thread1(PVOID arglist)
{
   APIRET rc;
   printf("... Posting semaphore\n");
   rc = DosPostEventSem(hevMyEvent);
}
```

Figure 10-7. *Example program that uses an event semaphore.*

```
Starting program
... Posting semaphore
Semaphore posted
... Posting semaphore
Semaphore posted
```

Figure 10-8. *Output from program in Figure 10-7.*

Although the above program is trivial, it points out the relationship between the wait and the post APIs. The event semaphore is initially set when created. Therefore, unless no post events occur, the first wait that is issued will wait until the post API is called. When the thread gets time to run and issue the post API, the main thread will have an opportunity to fall out and continue. When this occurs, it resets the event semaphore back to 0 posts and starts the process over again. Lastly, it closes the semaphore before exiting the program.

If the DosResetEventSem API is not called in this program, the second wait API that is called in the main thread will not have to wait since the semaphore is already posted by the first thread. This, potentially, could result in an error being returned by the post API since the DosCloseEventSem API may have time to complete before the post API is called.

MULTIPLE WAIT SEMAPHORES

Multiple wait (muxwait) semaphores are used when it is important to wait on several event or mutual exclusion semaphores. The two semaphore types cannot be mixed. In OS/2 2.1, there are seven APIs that are available to help manage multiple wait semaphores.

- ❑ DosCreateMuxWaitSem
- ❑ DosWaitMuxWaitSem
- ❑ DosCloseMuxWaitSem
- ❑ DosAddMuxWaitSem
- ❑ DosDeleteMuxWaitSem
- ❑ DosOpenMuxWaitSem
- ❑ DosQueryMuxWaitSem

To create a muxwait semaphore, you must already have created the event or mutex semaphores that your program needs to wait for. One other design consideration is whether you want to wait until all the semaphores are posted or released, or just wait for one of them.

DosCreateMuxWaitSem

```
          DosCreateMuxWaitSem(
                                        PSZ pszName,
                                        PHMUX phmux,
                                        ULONG cSemRec,
                                        PSEMRECORD pSemRec,
                                        ULONG flAttr);
```

Parameters:

- ❑ pszName (PSZ) input—The name for the semaphore. If NULL is passed, the semaphore is unnamed.
- ❑ phmux (PHMUX) output—A pointer to a storage location where the semaphore handle is returned.
- ❑ cSemRec (ULONG) input—The size of the array passed in pSemRec.
- ❑ pSemRec (PSEMRECORD) input—An array of semaphore record structures which contain the following fields:

  hSemCur—For storing the event or mutex semaphore handle.

  ulUser—For associating an identifier with this semaphore.

- ❑ flAttr (ULONG) input—Private or shared semaphore.

```
0  Private
1  Shared
```

Figure 10-9 is an example illustrating how to create a muxwait semaphore that is to wait for two event semaphores:

```
HEV hevMyEvent1 = (HEV)NULL;  /* Global variable where an     */
                              /* event sem is stored          */
HEV hevMyEvent2 = (HEV)NULL;  /* Global variable where an     */
                              /* event sem is stored          */
```

```
HMUX hmuxMyMuxwait = (HMUX)NULL; /* Global variable where a  */
                                 /* muxwait sem is stored    */

APIRET rc;                   /* Local variable where an API  */
                             /* return code is stored        */

SEMRECORD sr[2];

rc = DosCreateEventSem((PSZ)NULL,                /* Unnamed sem */
           &hevMyEvent1,     /* Address for semaphore handle */
           0,                           /* Private semaphore */
           FALSE);                      /* Initially set     */
rc = DosCreateEventSem((PSZ)NULL,       /* Unnamed sem       */
           &hevMyEvent2,     /* Address for semaphore handle */
           0,                           /* Private semaphore */
           FALSE);                      /* Initially set     */
sr[0].hsemCur = (HSEM)hevMyEvent1;
sr[0].ulUser = 1;
sr[1].hsemCur = (HSEM)hevMyEvent2;
sr[1].ulUser = 2;
rc = DosCreateMuxWaitSem((PSZ)NULL,
           &hmuxMyMuxwait,   /* Address for semaphore handle */
           2,                /* Count of semaphore array     */
           sr,               /* Address of sem array         */
           DCMW_WAIT_ALL);   /* Wait for all sems to post    */
```

Figure 10-9. *DosCreateMuxWaitSem introduction.*

The Figure 10-9 example creates a muxwait semaphore based on two event semaphores. The DCMW_WAIT_ALL option on the DosCreateMuxWaitSem API indicates that the wait should end only when both event semaphores are posted.

Now that the muxwait semaphore is created, the DosWaitMuxWaitSem API can be used to wait on the event that the semaphore was created for.

DosWaitMuxWaitSem

```
        DosWaitMuxWaitSem(
                          HMUX hmux,
                          ULONG ulTimeout,
                          PULONG pulUser);
```

Parameters:

- ❑ hmux (HMUX) input—Semaphore handle.

- ❑ ulTimeout (ULONG) input—The number of milliseconds to wait for the semaphores, or SEM_INDEFINITE_WAIT, which means to wait forever.
- ❑ pulUser (PULONG) output—The identifier associated with the event or mutex that caused the API to return.

Since this wait is based on the previous create, the DosWaitMuxWaitSem API will not return until both event semaphores have been posted. Like the mutual exclusion and event semaphores, the multiple wait semaphores allow the program to wait indefinitely for the event to occur or allow a time-out value to be passed. When the DosWaitMuxWaitSem API returns, it returns the value of the semaphore user identifier that caused the return.

The user identifier is passed in the ulUser field of the SEMRECORD structure during the DosCreateMuxWaitSem call. This is important if the muxwait semaphore was created with the DCMW_WAIT_ANY option instead of the DCMW_WAIT_ALL option, as it indicates which event occurred that caused the API to return.

When the semaphore is no longer required by the process, it should be closed. When all processes have terminated their use of the muxwait semaphore, you can close it, which returns the semaphore to the system.

DosCloseMusWaitSem

```
        DosCloseMuxWaitSem(
                              HMUX hmux);
```

Parameter:

- ❑ hmux (HMUX) input—Semaphore handle.

Occasionally you may be faced with needing to add or delete a semaphore from a muxwait's semaphore list. The DosAddMuxWaitSem and DosDeleteMuxWaitSem APIs are used to add and delete semaphores from an existing muxwait semaphore list. This gives the convenience of not having to know at the time the muxwait semaphore is created what the complete list needs to be.

Like mutex and event semaphores, the muxwait semaphores can be accessed by other processes. Any time after the DosCreateMuxWaitSem has been issued, another process can issue the DosOpenMuxWaitSem API to access it as if this process had created this semaphore.

For the DosOpenMuxWaitSem API to succeed, the semaphores in the muxwait list must be opened first. When the process that issued the DosOpenMuxWaitSem API is finished using the muxwait semaphore, it should issue the DosCloseMuxWaitSem API to indicate this. Only when all processes have finished closing the muxwait semaphore will the semaphore be returned to the system.

The DosQueryMuxWaitSem API is used to query a muxwait semaphore's list of event or mutex semaphores as well as the characteristics with which it was created.

Figure 10-10 is a program illustrating the use of a muxwait semaphore. This program was derived from the one written to illustrate mutual exclusion semaphores in Figure 10-5.

```
/*  Muxwait semaphore.                                                  */

#define INCL_DOSSEMAPHORES
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>

#define STACKSIZE 4096

VOID PrintABC(PVOID arglist);
HMTX hmtxMySem = (HMTX)NULL;
HMUX hmuxMyMuxwait;

SEMRECORD hevMyEvents[10];

INT main(VOID)
{
   ULONG ulUser;
   USHORT i;

   DosCreateMutexSem((PSZ)NULL, &hmtxMySem, 0, FALSE);
   memset(hevMyEvents, '\0', sizeof(hevMyEvents));
   for ( i = 0; i < 10; ++i) {
      DosCreateEventSem((PSZ)NULL,
                        (PHEV)&hevMyEvents[i].hsemCur,
                        0, FALSE);
      hevMyEvents[i].ulUser = i;
   }
   DosCreateMuxWaitSem((PSZ)NULL, &hmuxMyMuxwait, 10,
                         hevMyEvents, DCMW_WAIT_ALL);
   for ( i = 0; i < 10; ++i)
      _beginthread(PrintABC, NULL, STACKSIZE,
                   (PVOID)hevMyEvents[i].hsemCur);
   DosWaitMuxWaitSem(hmuxMyMuxwait, SEM_INDEFINITE_WAIT,
                     &ulUser);
   DosCloseMuxWaitSem(hmuxMyMuxwait);
   for ( i = 0; i < 10; ++i)
      DosCloseEventSem((HEV)hevMyEvents[i].hsemCur);
   DosCloseMutexSem(hmtxMySem);
   return(0L);
}

VOID PrintABC(PVOID arglist)
{
   DosRequestMutexSem(hmtxMySem, SEM_INDEFINITE_WAIT);
   printf("A");
   printf("B");
   printf("C ");
   DosReleaseMutexSem(hmtxMySem);
   DosPostEventSem((HEV)arglist);
}
```

Figure 10-10. *Program illustrating the use of a muxwait semaphore.*

Notice the difference between the program in Figure 10-10 and the one in Figure 10-5. Remember that the mutex example program used a DosSleep to arbitrarily decide when all threads should have had enough time to complete their processing and end.

This is a wasteful and potentially incorrect method to determine if this has happened. Instead of using the DosSleep API, the program in Figure 10-10 passes an event semaphore to the thread. At the end of the thread's processing, it posts the semaphore indicating to wait no longer for it.

A muxwait semaphore is created to wait for each event semaphore that is created for every thread. Once all the event semaphores are posted, the DosWaitMuxWaitSem API will return indicating that all the threads are finished using the semaphores and therefore all of the semaphores can be closed.

SEMAPHORE USAGE DESIGN CONCERNS

The examples that have been discussed in this chapter show the proper syntax for programming semaphores in a typical design. It is intended as a framework for your applications design. You must adhere to the fundamentals or else the result of your program's execution will be unspecified.

Simply put, your design must properly issue the create, wait or request, post or release, and close APIs for the semaphores that your program requires.

After you have chosen the type of semaphore you need, you should call the DosCreate*Sem API. If another process requires access to the same semaphore, the DosOpen*Sem API should be issued after the process creating the semaphore has issued the DosCreate*Sem API. Once done, either process can use the semaphore. If the DosOpen*Sem API is issued before the DosCreate*Sem API has been issued, the DosOpen*Sem API will return ERROR_SEM_NOT_FOUND.

An appropriate place to put the DosCreate*Sem API is in an initialization routine that is called during the beginning of the main() function. If your code is in a Dynamic Link Library, an appropriate place may be the _DLL_InitTerm function. This function is called the first time a process accesses your DLL.

In the full-program examples shown in this chapter, the semaphores have always been closed in the main() function just before exiting the program. This is a convenient time for removing any resource your program has allocated.

However, if your program has an execution exception, it will never execute this code. In this event, your program does not properly close its resources. The operating system will close semaphores that are open for a process that is closing. However, your program should close the semaphore at the appropriate time within your application. If this is at the end of your program, you should use either an exception handler or exit list processing.

Exception handlers are called when an abnormal condition exists during execution of the program. This topic is discussed further in Chapter 14. The DosExitList API can be used to register an exit list routine that is to be called when the program ends for any reason.

Just before the exit routine is called, the system transfers ownership to the thread in which the exit routine is executed. This allows your exit routine the capability to issue the request for a semaphore. More importantly, the exit routine gives you the opportunity to close your process's semaphores. Figure 10-11 is an example program using an exit list routine to close a semaphore.

```
/*  prog10f11.  Exit list for closing semaphores.                */

#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#include <os2.h>

HMTX hmtxMySem = (HMTX)NULL;

VOID APIENTRY CleanResources();

INT main(VOID)
{
   DosCreateMutexSem("\\SEM32\\MYSEMTEST", &hmtxMySem, 0,
                     FALSE);
   DosExitList(EXLST_ADD, (PFNEXITLIST)CleanResources);
   return(0L);
}

VOID APIENTRY CleanResources()
{
   DosCloseMutexSem(hmtxMySem);
   DosExitList(EXLST_EXIT, (PFNEXITLIST)CleanResources);
}
```

Figure 10-11. *Program illustrating the use of an exit list for closing semaphores.*

When you have made sure your program properly opens and closes its use of a semaphore, your attention can turn to the design for using the semaphore. There are a few situations that you should consider when designing your application's use of semaphores. The situations that should be considered are properly releasing the semaphore, managing the case when the semaphore owner dies, and preventing deadlock conditions.

The owner of a mutex semaphore is the thread that was last granted access to the semaphore (for example, successful return from the DosRequestMutexSem API). The owner thread is allowed to nest requests for the semaphore. The owner, therefore, must also release each of the requests that were issued.

If one less release is issued, the thread remains the owner. The use of semaphores in a complicated, even recursive, program will increase the likelihood that problems of this nature will occur. To help isolate such a problem or to avoid it, you might consider using something like the function in Figure 10-12 when you know your thread is finished with the resource.

```
VOID ReleaseMutex(HMTX thishmtx)
{
   ULONG count;
   PID processID;
   TID threadID;
   DosQueryMutexSem(thishmtx, &processID, &threadID,
                    &count);
   if (count > 1) {
      /* log an error                                              */
   }
   for (; count > 0; count--) {
      DosReleaseMutexSem(thishmtx);
   }
}
```

Figure 10-12. *Function that completely releases a mutex semaphore.*

The ReleaseMutex function in Figure 10-12 asks how many DosRequestMutexSem APIs have been called without a call to DosReleaseMutexSem and then issues that number of them. Techniques such as this can help isolate or prevent implementation problems.

Another design point to consider is how to handle the case when the semaphore owner dies. If the thread that owns the semaphore dies, another thread that was waiting for the semaphore to be released will receive the ERROR_SEM_OWNER_DIED return code. When receiving this error, the resource should be considered to be in an unusable state and should not be accessed. This is more likely to happen with shared semaphores used across processes. In this case, the processes should manage this event by closing its use of the semaphore.

Another design point to consider is what a thread is doing while it is the owner of a semaphore. If the owning thread requests access to a different semaphore, the possibility for deadlock exists. A deadlock condition exists when two threads both own different semaphores and are attempting to request the ownership of the other thread's semaphore. In this situation, neither thread will ever get access. If the SEM_INDEFINITE_WAIT option is used, both threads will wait forever.

The program in Figure 10-13 illustrates this condition.

```
/*  prog10f13.   Potential deadlock.                             */

#define INCL_DOSSEMAPHORES
#define INCL_DOSPROCESS
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>

#define STACKSIZE 4096
VOID Thread2(PVOID arglist);
VOID Thread3(PVOID arglist);
HMTX hmtxScreen = (HMTX)NULL,
     hmtxArray = (HMTX)NULL;
USHORT myarray[10];
INT main(VOID)
{
   USHORT i;

   DosCreateMutexSem((PSZ)NULL, &hmtxScreen, 0, FALSE);
   DosCreateMutexSem((PSZ)NULL, &hmtxArray, 0, FALSE);
   memset(myarray, '\0', sizeof(myarray));
   _beginthread(Thread2, NULL, STACKSIZE, NULL);
   _beginthread(Thread3, NULL, STACKSIZE, NULL);
   DosSleep(5000);
   DosCloseMutexSem(hmtxScreen);
   DosCloseMutexSem(hmtxArray);
   return(0L);
}

VOID Print0to9()
{
   USHORT i;

   for (i=0; i < 10; ++i) printf("i = %d ", i);
   printf("\n");
}

VOID UpdateArray()
{
   USHORT i;

   for (i=0; i < 10; ++i) myarray[i]++;
}

VOID Thread2(PVOID arglist)
{
   USHORT i;

   for (i=0; i < 10; ++i) {
      DosRequestMutexSem(hmtxScreen, SEM_INDEFINITE_WAIT);
      Print0to9();
      DosRequestMutexSem(hmtxArray, SEM_INDEFINITE_WAIT);
      UpdateArray();
      DosReleaseMutexSem(hmtxArray);
```

```
         DosReleaseMutexSem(hmtxScreen);
      }
   }

   VOID Thread3(PVOID arglist)
   {
      USHORT i;

      for (i=0; i < 10; ++i) {
         DosRequestMutexSem(hmtxArray, SEM_INDEFINITE_WAIT);
         UpdateArray();
         DosRequestMutexSem(hmtxScreen, SEM_INDEFINITE_WAIT);
         Print0to9();
         DosReleaseMutexSem(hmtxArray);
         DosReleaseMutexSem(hmtxScreen);
      }
   }
```

Figure 10-13. *A program illustrating a potential deadlock situation.*

Notice in the program in Figure 10-13 that there are two semaphores. One is to protect the screen and the other is to protect a buffer. Thread2 accesses the screen semaphore first and Thread3 accesses the array semaphore first. Both threads properly access the semaphores and use the resource they protect.

However, notice that Thread3 releases the array semaphore before the screen semaphore. Because of this, the program may eventually get into a state where both threads are blocked forever. The design must consider the case in which a thread needs access to a semaphore while it is the owner of one.

Before concluding this discussion of OS/2 semaphores, note the importance of three additional semaphore APIs that are available for Presentation Manager programs:

- ❑ WinRequestMutexSem is a replacement for DosRequestMutexSem.
- ❑ WinWaitEventSem is a replacement for DosWaitEventSem.
- ❑ WinWaitMuxWaitSem is a replacement for DosWaitMuxWaitSem.

The PM APIs are for programs that use the Presentation Manager interface. They allow the processing of messages sent by means of the WinSendMsg API to windows created on the same thread that is calling the semaphore API.

If you use the DOS version of the same API, the thread sending the message will be blocked until the semaphore API returns and the thread returns to a state in which it can accept asynchronous messages.

Therefore, if you don't use the PM versions of these semaphore APIs, you increase the possibility of encountering deadlock conditions.

SUMMARY

Semaphores are a key tool used by multiple threaded applications to properly protect resources and signal other threads about events occurring. This chapter has discussed the following categories of OS/2 2.1 semaphores:

Mutual Exclusion (Mutex) Semaphores

Mutex semaphores are primarily used to serialize access to a resource. The following APIs are used to manage a mutex semaphore.

- ❑ DosCreateMutexSem—This API is called to create a mutex semaphore. After the semaphore has been created, the other Dos*MutexSem APIs can be called.

- ❑ DosRequestMutexSem—This API is used to request ownership of a mutex semaphore. When this API returns successfully, the semaphore will be owned by the calling thread.

- ❑ DosReleaseMutexSem—This API is used to release an owned mutex semaphore. Once the semaphore is released, any thread can request the semaphore.

- ❑ DosCloseMutexSem—This API is used to close the process's use of a mutex semaphore. When all processes that have created or opened a mutex semaphore have closed it, the semaphore will be returned to the operating system.

- ❑ DosOpenMutexSem—This API is used to open access to a mutex semaphore that was created in another process. Once access to the semaphore is granted, the DosRequestMutexSem API can be called.

- ❑ DosQueryMutexSem—This API is used to query the basic characteristics about the owner of a mutex semaphore.

Event Semaphores

Event semaphores are primarily used to notify another thread that an event has occurred. The following APIs are used to manage an event semaphore:

- ❑ DosCreateEventSem—This API is used to create the event semaphore. When the semaphore has been created, the other Dos*EventSem APIs can be called.

- ❑ DosWaitEventSem—This API is used to wait for an event semaphore until another thread issues the DosPostEventSem API. Once the post occurs, this API will return successfully.

- ❑ DosPostEventSem—This API is used to indicate that an event has occurred. The act of posting the semaphore will wake any thread that was waiting for this event to occur.

- ❑ DosCloseEventSem—This API is used to close a process's use of an event semaphore. When all processes that have opened or created the event semaphore have closed it, the semaphore will be returned to the operating system.

- ❑ DosOpenEventSem—This API is used to gain access to an event semaphore from a process other than the one that created it. When access is granted, other event semaphore APIs can be used against it.

- ❑ DosQueryEventSem—This API is used to query the number of times an event semaphore has been posted.

- ❑ DosResetEventSem—This API is used to reset an event semaphore so that the wait and post cycle can begin again. Once this API has completed, and the event semaphore has been completely reset, a thread can wait on the semaphore to be posted again.

Multiple Wait (Muxwait) Semaphores

Muxwait semaphores are used when it is necessary to wait for more than one event or mutex semaphore. The following APIs are used to manage a muxwait semaphore.

- ❑ DosCreateMuxWaitSem—This API is used to create a muxwait semaphore based on a list of existing event and mutex semaphores. Once the semaphore has been created, the other Dos*MuxWaitSem APIs can be called.

- ❑ DosWaitMuxWaitSem—This API is used to wait for a muxwait semaphore until one or all of the multiple semaphores have been cleared. A semaphore is cleared when an event semaphore is posted or when a mutex semaphore is released.

- ❑ DosCloseMuxWaitSem—This API is used to close a process's use of a muxwait semaphore. Once all processes that have opened or created the muxwait semaphore have closed it, the semaphore will be returned to the operating system.

- ❑ DosAddMuxWaitSem—This API is used to add an event or mutex semaphore to an existing muxwait semaphore's list.

- ❑ DosDeleteMuxWaitSem—This API is used to delete an event or mutex semaphore from an existing muxwait semaphore's list.

- ❑ DosOpenMuxWaitSem—This API is used to gain access to a muxwait semaphore from a process other than the one that created it. Once access is granted, the other muxwait semaphore APIs can be used on it.

- ❑ DosQueryMuxWaitSem—This API is used to query the list of event and mutex semaphores associated with a muxwait semaphore.

CHAPTER 11

Pipes

"Therefore, ye soft pipes, play on . . . ." —*Keats*

INTRODUCTION

You have seen that OS/2 2.1 is a very powerful and flexible multitasking system. You can run several programs simultaneously, and it is not too difficult to imagine wanting to have some of those programs communicate with each other.

Sometimes the programs will be designed to work together, as with a communications protocol. Other times, the programs will not be designed to work together. For example, you may want to take information from a database and put it into a spreadsheet, or you may want to call a compiler and display the results from within an editor.

PIPES AS AN INTERPROCESS COMMUNICATION METHOD

OS/2 2.1 provides several mechanisms for interprocess communication (IPC), with the main IPC objects being pipes, queues, semaphores, and shared memory. This chapter looks at pipes.

Pipes are probably the most powerful of the IPC mechanisms. A pipe is a system-controlled buffer used to relay information between processes. The information can take any form, from text to binary. The size of the pipe determines how much data can be sent at one time and how much data can be stored while it is waiting to be read. You specify the size of the pipe when it is created; the system may allocate less if not enough memory is available. You can specify which way data flows through the pipe, or you can make it duplex (two-way).

The system ensures that no data is overwritten before it is read. Pipes and files use the same type of handles, and pipes can be accessed using DosRead and DosWrite. See Chapter 6 for examples of the DosRead and DosWrite functions. Before getting into the details, you'll want to compare the different types of IPC objects to see when each would be useful.

PIPES AND SEMAPHORES

You have seen that semaphores are used to synchronize processes. Pipes can also be used for synchronization in that a process can wait for a message to arrive through the pipe. However, when all that is really needed is a stop orgo signal, semaphores are the logical choice. Semaphores are also the logical choice when you are working with multiple events and waiting for any one of them, or all of them, to happen.

On the other hand, if you want to pass data from one process to another, then a semaphore alone will not be sufficient. Furthermore, if you want to pass data from a database to a spreadsheet, it may be that neither of those is written to wait on a semaphore. Even though neither application is written to use a pipe, both can still be made to do so, without having to recompile either one. Lastly, semaphores cannot be accessed across a network; pipes can.

PIPES AND QUEUES

Queues, like pipes, allow you to pass data between processes. However, queues only allow you to pass 32 bits of data. This means that with queues, you can pass either flags or pointers, not both. To pass as much actual data as pipes do, you would have to allocate shared memory and pass pointers to it. Another difference is that while you can specify which end of a pipe can read from it, only the owner of a queue can read from it. Queues give you the ability to prioritize messages. Lastly, as with semaphores, queues cannot be accessed across a network; pipes can.

PIPES AND SHARED MEMORY

Both pipes and shared memory are used to pass information between processes. With shared memory, a process can overwrite data; with pipes it cannot. With shared memory the programmer must keep track of which data has been read; with pipes the system does this. In this version of OS/2, memory cannot be shared across a network. Finally, pipes can be used to imitate files, using DosRead and DosWrite APIs.

TYPES OF PIPES

OS/2 2.1 provides two types of pipes: *named* and *unnamed*. Unnamed pipes can be used only between related processes, that is, when one process starts the other with DosExecPgm. Named pipes can be used by any process that knows the name. All pipes have multiple handles. An unnamed pipe has two distinct handles per process, a read handle and a write handle, just as files do. A named pipe has one distinct handle per process, which is used for both reading and writing. If the process created the pipe, then the handle is called a *server end* handle; otherwise, it is called a *client end* handle. Named pipes have a great deal of flexibility; unnamed pipes are not as flexible but can transparently replace a child process's stdin (for example, the keyboard) and stdout (for example, the display).

UNNAMED PIPES

Unnamed pipes are used to pass data between related processes. Using an unnamed pipe is a little like playing a trumpet. Anyone who uses it blows air into one side and hears the result from the other side. The data is always read and written as a simple byte stream, with nothing added by the system.

Unnamed pipes are opened with DosCreatePipe, closed with DosClose, and accessed with DosRead and DosWrite. Thus, they can replace stdin, stdout, or stderr for a process almost transparently. This makes them ideal to use to replace a child process's stdin or stdout. Figure 11-1 shows how to make a child process's stdin a pipe instead of the keyboard, without losing the parent's original stdin.

The program in Figure 11-1 starts a child process that executes OS/2 2.1 command line commands. By replacing its stdin with a pipe, you cause the child process to accept commands from your program instead of from the keyboard. The example passes the dir command, so that the child process will display the contents of the current directory.

Figure 11-1 introduces the DosCreatePipe API.

DosCreatePipe

```
        DosCreatePipe(
                     PHFILE readhandle,
                     PHFILE writehandle,
                     ULONG  pipesize);
```

Parameters:

- ❑ readhandle (HFILE) output—The call returns two handles. This parameter is where to put the read handle. Pass the address of readhandle.
- ❑ writehandle (HFILE) output—Where to put the write handle. Pass the address of writehandle.
- ❑ pipesize (ULONG) input—How big to make the pipe.

```
/*   Replacing stdin of a child process with an unnamed pipe.       */

#define INCL_DOSQUEUES
#define INCL_DOSFILEMGR
#define INCL_DOSPROCESS
#define PIPESIZE  4096
#define OBJBUFLEN  100
#include <os2.h>

int main (int argc, void *argv[])
{

   /* The next variables are for the pipe calls                     */
   HFILE MyIn     = 0;              /* Handle of original std in   */
   HFILE SaveMyIn = -1;             /* Handle for saving std in    */
   HFILE PipeRead;                  /* Read handle for the pipe    */
   HFILE PipeWrite;                 /* Write handle for the pipe   */
   APIRET rc;
   /* The next variables are for the DosExecPgm call                */
   char  ObjNameBuf[OBJBUFLEN];
   ULONG ExecFlags = EXEC_ASYNC;
   RESULTCODES kidrc;
   ULONG       byteswritten = 0;
   ULONG       bytesread    = 0;
   char        buf[PIPESIZE];

   memset(&buf, '\0', PIPESIZE);           /* Initialize the buffer  */
   DosDupHandle(MyIn, &SaveMyIn);          /* Save original std in   */

   rc = DosCreatePipe(&PipeRead,           /* Read Handle            */
                      &PipeWrite,          /* Write Handle           */
                      PIPESIZE);           /* pipe size              */

   DosDupHandle(PipeRead, &MyIn);          /* Replaces std in        */
   /* Start the child process */
   rc = DosExecPgm(ObjNameBuf,             /* Buffer if failure      */
        OBJBUFLEN,
        ExecFlags,                         /* Asynch operation       */
        NULL,                              /* child has no parms     */
        NULL,                              /* Inherit environment    */
        &kidrc,
        "HIWORLD.EXE");                    /* Name of child pgm      */
       /* Child inherits current std in and std out                  */
       /* Current std in is the unnamed pipe                         */

   DosDupHandle(SaveMyIn, &MyIn)           /* Restore orig std in    */

   /* Now send input to the child process                            */
   rc = DosWrite(PipeWrite,                /* Child's stdin          */
        "Hello, world!\n",                 /* data                   */
        14,                                /* length of data         */
        &byteswritten);

   DosSleep(5000);                         /* Let child complete     */
   DosClose(PipeRead);                     /* Close the pipe         */
   DosClose(PipeWrite);
}

/* This is the code for HIWORLD.C                                    */
#include <os2.h>
#include <stdio.h>

main(argc, argv, envp)
   int argc;
   char *argv[];
   char *envp[];
{
   char buf1[100];
   scanf("%[^\n]", buf1);
   printf("%s\n", buf1);
}
```

Figure 11-1. *Replacing a child process's stdin with an unnamed pipe.*

Using a pipe instead of a file is not completely transparent. The pipe's existence in memory instead of on a disk or other storage device is transparent, and (except for the open) the same APIs can be used. There are two main differences: First, the return codes are different, and, second, a pipe has two handles instead of one. This is not a problem. When used to replace stdin, only the read handle will ever be needed. Similarly, only the write handle is needed to replace stdout. Otherwise, both processes know they are using a pipe.

Combining Pipes and Files

One of the examples mentioned earlier was the desire to start a compiler/linker from within a text editor and to display the results from within the text editor. For the sake of this example, assume that the input commands for the compiler/linker are in a file. The next step is for the text editor to make the compiler read commands from the file and send its results back to the text editor. To do this, set the compiler's stdin to be the file and the compiler's stdout to be an unnamed pipe. Code for this is shown in Figure 11-2.

```
/*Replace stdin of a child process with a file and stdout with pipe */

#define INCL_DOSQUEUES
#define INCL_DOSFILEMGR
#define INCL_DOSPROCESS
#define PIPESIZE  4096
#define OBJBUFLEN  100
#include <os2.h>

int main (int argc, void *argv[])
{

   HFILE MyIn      = 0;              /* Handle of original std in   */
   HFILE SaveMyIn = -1;              /* Handle for saving std in    */
   HFILE MyOut     = 1;              /* Handle of original std out  */
   HFILE SaveMyOut= -1;              /* Handle for saving std out   */
   HFILE PipeRead;                   /* Read handle for the pipe    */
   HFILE PipeWrite;                  /* Write handle for the pipe   */
   HFILE FileReadWrite;              /* Handle for a file           */
                                     /* -- A file only has 1 handle */
   APIRET rc;

   /*   The next variables are for the DosExecPgm call              */
   char  ObjNameBuf[OBJBUFLEN];
   ULONG ExecFlags = EXEC_ASYNC;
   RESULTCODES kidrc;
   ULONG       byteswritten = 0;
   ULONG       bytesread    = 0;
   char        buf[PIPESIZE];
```

```
/*    The next variables are for the DosOpen call                   */
ULONG    Action;
ULONG    OpenFlag = OPEN_ACTION_FAIL_IF_NEW |
                    OPEN_ACTION_OPEN_IF_EXISTS;
ULONG    OpenMode = OPEN_FLAGS_SEQUENTIAL  |
                    OPEN_FLAGS_NOINHERIT   |
                    OPEN_SHARE_DENYWRITE   |
                    OPEN_ACCESS_READONLY;

memset(&buf, '\0', PIPESIZE);              /* Initialize the buffr*/
DosDupHandle(MyIn, &SaveMyIn);             /*Save original std in */
DosDupHandle(MyOut, &SaveMyOut);           /*Save original std out*/

/* Prepare to replace the child's std in with the file:            */
/* Open the file, since stdin is always initially open             */
rc = DosOpen("d:\\myprogs\\aninfile",        /* Name of the file  */
     &FileReadWrite,                         /* File handle       */
     &Action,                                /* output status     */
     0,                                      /* ignored           */
     0,                                      /* ignored           */
     OpenFlag,
     OpenMode,
     0);                                     /* No extended attrs */

DosDupHandle(FileReadWrite, &MyIn);           /* Replace std in    */

                        /* Prepare to replace the child's std out*/
                        /* with a pipe:                            */
DosCreatePipe(&PipeRead,                    /* Create the pipe    */
              &PipeWrite,
              PIPESIZE);
DosDupHandle(PipeWrite, &MyOut);            /* Replace std out    */

/* Start the child process                                         */
DosExecPgm(ObjNameBuf,                     /* Buffer if failure   */
     OBJBUFLEN,
     ExecFlags,                            /* Asynch operation    */
     NULL,                                 /* child has no parms  */
     NULL,                                 /* Inherit environment */
     &kidrc,
    "HIWORLD.EXE");                        /* Name of child pgm   */
        /* Child inherits current std in and std out               */
        /* Current std in is the file                              */
        /* Current std out is the unnamed pipe                     */

                /* Restore stdin and stdout                        */
DosDupHandle(SaveMyIn,  &MyIn);
DosDupHandle(SaveMyOut, &MyOut);

DosSleep(2000);
/* Process child's results                                         */
DosRead(PipeRead, buf, PIPESIZE, &bytesread);
if (bytesread != 0) {
   printf("%s", buf);
   }
   DosClose(PipeRead);
   DosClose(PipeWrite);
}
```

Figure 11-2*. Replacing a child process's stdin with a file and replacing its stdout with a pipe.*

Having Children Talk to Each Other

Anyone with children knows how difficult it can be to maintain peace. With unnamed pipes, it is easy to get children to talk to each other.

Suppose you have two chess programs that you want to play each other. One program will make a move, the other program will respond, and so on. You could tie them together by reading output from the first and passing it to the second, and vice versa (see Figure 11-3), or you can use unnamed pipes to have them talk directly to each other (see Figure 11-4). Notice that in both figures, the chess programs think they are accessing the keyboard and display, even though they are not.

```
/*********************************************************************/
/*     The naming convention used for unnamed pipes:                 */
/*     If the handle starts with a C, a child uses it                */
/*     If the handle starts with a P, the parent uses it             */
/*     C1 = child1, C2 = child2, ...                                 */
/*     F1 = file1,  F2 = file2,  ...                                 */
/*     R = read, W = write                                           */
/*                                                                   */
/*     For example, if a child is to read from its parent,           */
/*     then the handle for its stdin will be C1RPW.                  */
/*     The parent's handle for the parent's end of the pipe          */
/*     will be PWC1R.                                                */
/*                                                                   */
/* Two pipes are not sufficient for this example.  If a              */
/* child processs's std in is one end of an unnamed pipe             */
/* and its std out is the other end, then it can  read               */
/* the same data it wrote.                                           */
/* If semaphores are used to avoid this problem, then two            */
/* pipes would be sufficient.                                        */
/*********************************************************************/

#define INCL_DOSQUEUES
#define INCL_DOSFILEMGR
#define INCL_DOSPROCESS
#define PIPESIZE  4096
#define OBJBUFLEN  100
#include <os2.h>
```

```
int main (int argc, void *argv[])
{

   HFILE MyIn     = 0;              /* Handle of original std in   */
   HFILE SaveMyIn = -1;             /* Handle for saving std in    */
   HFILE MyOut    = 1;              /* Handle of original std out  */
   HFILE SaveMyOut= -1;             /* Handle for saving std out   */
   HFILE C1RPW;                     /* Read handle for pipe  1     */
   HFILE PWC1R;                     /* Write handle for pipe 1     */
   HFILE PRC1W;                     /* Read handle for pipe  2     */
   HFILE C1WPR;                     /* Write handle for pipe 2     */
   HFILE C2RPW;                     /* Read handle for pipe  3     */
   HFILE PWC2R;                     /* Write handle for pipe 3     */
   HFILE PRC2W;                     /* Read handle for pipe  4     */
   HFILE C2WPR;                     /* Write handle for pipe 4     */

   ULONG      byteswritten = 0;
   ULONG      bytesread    = 0;
   char       buf[PIPESIZE];

   APIRET rc;
   /* The next variables are for the DosExecPgm call              */
   char  ObjNameBuf[OBJBUFLEN];
   ULONG ExecFlags = EXEC_ASYNC;
   RESULTCODES kidrc;

   DosDupHandle(MyIn, &SaveMyIn);          /*Save original std in */
   DosDupHandle(MyOut, &SaveMyOut);        /*Save original std out*/

                     /* Create four pipes for the communication  */
   DosCreatePipe(&C1RPW,                   /* PIPE 1, read  handle*/
                 &PWC1R,                   /* PIPE 1, write handle*/
                 PIPESIZE);
   DosCreatePipe(&PRC1W,                   /* PIPE 2              */
                 &C1WPR,
                 PIPESIZE);
   DosCreatePipe(&C2RPW,                   /* PIPE 3              */
                 &PWC2R,
                 PIPESIZE);
   DosCreatePipe(&PRC2W,                   /* PIPE 4              */
                 &C2WPR,
                 PIPESIZE);

                         /* Setup Child1                          */
   DosDupHandle(C1RPW, &MyIn);              /* Replace std in     */
   DosDupHandle(C1WPR, &MyOut);             /* Replace std out    */

                         /* Start the child process               */
   rc = DosExecPgm(ObjNameBuf,             /* Buffer if failure   */
        OBJBUFLEN,
        ExecFlags,                         /* Asynch operation    */
        NULL,                              /* child has no parms  */
        NULL,                              /* Inherit environment */
        &kidrc,
        "HIWORLD.EXE");                         /* Name of child pgm    */

                          /* Setup Child2                                */
   DosDupHandle(C2RPW, &MyIn);                  /* Replace std in       */
   DosDupHandle(C2WPR, &MyOut);                 /* Replace std out      */

                          /* Start the child process                     */
   rc = DosExecPgm(ObjNameBuf,                  /* Buffer if failure    */
        OBJBUFLEN,
        ExecFlags,                              /* Asynch operation     */
        NULL,                                   /* child has no parms   */
        NULL,                                   /* Inherit environment  */
        &kidrc,
        "HIWORLD2.EXE");                        /* Name of child pgm    */

                          /* Restore stdin and stdout                    */
   DosDupHandle(SaveMyIn,  &MyIn);
   DosDupHandle(SaveMyOut, &MyOut);

                          /* Send input to child1                        */
   rc = DosWrite(PWC1R,
        "Hello, world\n",
        14,
        &byteswritten);

   /* In this case, both programs read and write once                    */
   DosSleep(1000);
   rc = DosRead (PRC1W,                       /* Read data from child1 */
        &buf,                                 /* data                   */
        PIPESIZE,                             /* max length of data     */
        &bytesread);
   printf("we read ---- %s\n", buf);
   DosWrite(PWC2R,                            /* Write it to child2     */
        &buf,                                 /* data                   */
        bytesread,                            /* length of data         */
        &byteswritten);

   DosRead(PRC2W,                             /* Read data from child2 */
        &buf,                                 /* data                   */
        PIPESIZE,                             /* max length of data     */
        &bytesread);
   printf("we read ---- %s\n", buf);
   DosWrite(PWC1R,                            /* Write it to child1     */
        buf,                                  /* data                   */
        bytesread,                            /* length of data         */
        &byteswritten);

   DosClose(PRC1W);
   DosClose(PRC2W);
   DosClose(PWC1R);
   DosClose(PWC2R);
   DosClose(C1RPW);
   DosClose(C1WPR);
   DosClose(C2RPW);
   DosClose(C2WPR);

}

      /* This is the code for HIWORLD2.C                              */
#include <os2.h>
#include <stdio.h>

main(argc, argv, envp)
   int argc;
   char *argv[];
   char *envp[];
{
   char buf1[100];
   scanf("%[^\n]", buf1);
   printf("My input was:%s\n", buf1);
}
```

Figure 11-3. *Passing information from one child to another, through the parent.*

```
/*  Two children reading and writing.                                 */

#define INCL_DOSQUEUES
#define INCL_DOSFILEMGR
#define INCL_DOSPROCESS
#define PIPESIZE  4096
#define OBJBUFLEN  100
#include <os2.h>

int main (int argc, void *argv[])
{

   HFILE MyIn     = 0;             /* Handle of original std in   */
   HFILE SaveMyIn = -1;            /* Handle for saving std in    */
   HFILE MyOut    = 1;             /* Handle of original std out  */
   HFILE SaveMyOut= -1;            /* Handle for saving std out   */
   HFILE Child1sInput;             /* Read handle for pipe 1      */
   HFILE Child2sOutput;            /* Write handle for pipe 1     */
   HFILE Child2sInput;             /* Read handle for pipe 2      */
   HFILE Child1sOutput;            /* Write handle for pipe 2     */
   ULONG byteswritten;
   APIRET rc;

   /*   The next variables are for the DosExecPgm call               */
   char  ObjNameBuf[OBJBUFLEN];
   ULONG ExecFlags = EXEC_ASYNC;
   RESULTCODES kidrc;

   DosDupHandle(MyIn, &SaveMyIn);          /*Save original std in */
   DosDupHandle(MyOut, &SaveMyOut);        /*Save original std out*/
             /* Create two pipes for the communication            */
   DosCreatePipe(&Child1sInput,                        /* PIPE 1   */
             &Child2sOutput,
             PIPESIZE);
   DosCreatePipe(&Child2sInput,                         /* PIPE 2   */
                 &Child1sOutput,
                 PIPESIZE);

                /* Set up Child1                                    */
  DosDupHandle(Child1sInput, &MyIn);         /* Replace stdin       */
   DosDupHandle(Child1sOutput,&MyOut);       /* Replace stdout      */

                /* Start the child process                          */
   rc = DosExecPgm(ObjNameBuf,               /* Buffer if failure   */
        OBJBUFLEN,
        ExecFlags,                           /* Asynch operation    */
        NULL,                                /* child has no parms  */
        NULL,                                /* Inherit environment */
        &kidrc,
        "HIWORLD.EXE");                      /* Name of child pgm   */

                /* Setup Child2                                     */
   DosDupHandle(Child2sInput, &MyIn);        /* Replace stdin       */
   DosDupHandle(Child2sOutput,&MyOut);       /* Replace stdout      */

                /* Start the child process                          */
   rc = DosExecPgm(ObjNameBuf,               /* Buffer if failure   */
        OBJBUFLEN,
        ExecFlags,                           /* Asynch operation    */
        NULL,                                /* child has no parms  */
        NULL,                                /* Inherit environment */
        &kidrc,
        "HIWORLD2.EXE");                     /* Name of child pgm   */

                /* Restore stdin and stdout                         */
   DosDupHandle(SaveMyIn,  &Child2sInput);
   DosDupHandle(SaveMyOut, &Child2sOutput);


   /* Send input to child1                                          */
   rc = DosWrite(Child1sInput,
        "Hello, world\n",
        14,
        &byteswritten);


                   /* Continue with parent program                  */
}
```

***Figure 11-4**. Setting two children to read and write to each other.*

NAMED PIPES

Named pipes are much more flexible than unnamed pipes. They can be used in many contexts. Processes do not have to be related; in fact, they do not even have to be on the same machine. Because they are created by one process and all other processes must attach to them to use them, named pipes are useful in a client-server environment.

Throughout this chapter the creator of the pipe is called the *server*, and any process that later attaches to it is called the *client*. You can create a variety of pipes.

A Smorgasbord of Named Pipes

Unlike unnamed pipes which are always bi-directional, a named pipe can also be uni-directional in either direction. An *inbound* pipe is one in which the creator, or server, can only read data and the client can only write data. An *outbound* pipe is one in which the server can only write data and the client can only read data. A *duplex* pipe allows both the server and the client to read and write. For a duplex pipe, OS/2 2.1 will keep track of who wrote what, so that a process never reads the same data that it wrote.

A named pipe can be either *blocking* or *nonblocking*. A blocking pipe is useful when you want a program to receive input from another program and not continue until it gets the input. For example, a server might wait for input from a client. This should be reminiscent of an event semaphore, except in this case data is exchanged, and OS/2 2.1 handles the requirements to wait, post, and reset.

A nonblocking pipe is useful when you want to continue regardless. For example, you want the process to respond to user input every two seconds but also be able to accept piped input every half-second.

By default, a read will block if the pipe is empty, and a write will block if the pipe is full. This can be changed so that a read will return ERROR_NO_DATA if the pipe is empty, and a write will return with a value of 0 for the bytes written parameter if the pipe is full (or if there is not enough room in the pipe for the entire message). This option is set when the pipe is created and can be changed at any time by either the server or the client by using the DosSetNPHState API.

A pipe can be blocking at one end and nonblocking at the other end; DosSetNPHState changes the mode only for that end of the pipe. The blocking mode for the server is initially set when the pipe is created. The blocking mode for the client is always set to blocking at first.

A named pipe can send data in two different formats, as bytes or as messages. If it uses the message format, it is called a *message pipe*. A message pipe must be a named pipe since unnamed pipes always send data as bytes.

In a message pipe, OS/2 2.1 adds a header for each write. Except for when a message is larger than the pipe, it does not allow partial messages to be written. If there is not enough room in the pipe for a message, the writer is blocked until the entire message has been written.

If a message is larger than the size of the pipe, the writer will be blocked until the reader has read enough to make room for the rest of the message. This last blocking action will occur even if the pipe is set to be nonblocking. If you want to guarantee that the writer will not be blocked, you must compare the size of the message to the size of the pipe before sending it.

A message pipe can be read as either messages or bytes. When a message pipe is read as bytes, you specify how many bytes you want to read. You will be returned that many bytes, skipping over the system-supplied headers and ignoring message boundaries, unless the pipe does not contain enough data. In that case, you will be given all the data in the pipe, and the bytes read parameter is set to the number of bytes actually read.

When a message pipe is read as messages, you supply a buffer for the message. If the buffer is not large enough to hold the next message, then the buffer will be filled and the return code set to ERROR_MORE_DATA. You will not be able to read any more from the pipe until the rest of that message can be read. The read mode is set when the pipe is created and can be changed at any time, by either the server or the client, with the DosSetNPHState API.

Just as a pipe can be blocking at one end and nonblocking at the other end, a pipe can be read as bytes at one end and messages at the other end. DosSetNPHState only changes the mode for that end of the pipe. The read mode for the server is initially set when the pipe is created. The read mode for the client is always initially set to bytes. This is shown in Table 11-1.

	SERVER	CLIENT
Read	X	Bytes
Write	X	X
Blocking	X	Blocking
Note: X = the mode specified in the DosCreateNPipe API when the pipe was created.		

Table 11-1. *Initial settings for named pipes.*

When both ends of the pipe are on the same system, data that is written is immediately available to be read. When one end is remote, the operating system may want to buffer the writes locally and send them across the network at a later time. This would mean that a reader might block, or get ERROR_NO_DATA, even though data had been written to the pipe. When the pipe is created, you can allow local buffering of writes by specifying NP_WRITEBEHIND. You can disallow it, that is, force the write to be sent across the network immediately, by specifying NP_NOWRITEBEHIND. Since unnamed pipes cannot be remote, this is not an issue for them.

Another feature of named pipes is that you can have several instances of the same pipe active at the same time. That is, you can create several pipes with the same name. Each instance would have a unique handle for each end. Multiple instances are useful when you want a single server to be able to respond to several clients simultaneously. When you create a pipe for the first time, you specify how many instances can be created. This number cannot be changed when you are creating future instances. The section later in this chapter, *Coordinating Multiple Instances with Semaphores,* contains more information on how to use multiple instances efficiently.

Creating and Using a Named Pipe

The server and client access the pipe a little differently. The server first creates a pipe using DosCreateNPipe. Then it must connect to it using DosConnectNPipe. At this point, it must wait for a client to attach to the pipe. After that happens, the server can use DosRead and DosWrite to receive and send data through the pipe. When the communication is through, the server should wait for the client to detach from the pipe and then issue DosDisconnectNPipe. Now the server can either connect to the pipe again and wait for another client, or it can discard the pipe using DosClose.

If there are multiple instances of the pipe, the server connects and accesses each instance individually. A pipi can be simultaneously connected to one instance and disconnected from another instance. The server distinguishes between them by means of the handles.

The client attaches to the pipe using DosOpen. If the server is connected and no other client is attached, the client will attach. Otherwise, an ERROR_PIPE_BUSY will be returned, and the client can wait for the pipe to be accessible by using DosWaitNPipe. Once the pipe is accessible, it must issue a DosOpen again. Once the client is attached, it uses DosRead and DosWrite just like the server. To detach, the client only issues DosClose.

In general, a client will be attached to only one instance of the pipe. If any instances are available when the client does a DosOpen, exactly one of them will be attached. When both ends of the pipe are closed, OS/2 2.1 will reclaim any resources allocated to the pipe. The server would have to do another DosCreateNPipe before it or a client could connect to it again.

Figure 11-5 shows the server and client sides for creating and accessing a named pipe. Notice two things about Figure 11-5. First, when the pipe is created, a timeout is specified so that clients waiting to attach to a pipe do not have to wait forever. This timeout can be overridden on the DosWaitNPipe call. Second, the pipe is created in blocking mode. If the pipe is in nonblocking mode when the server issues a DosConnectNPipe, the call will fail immediately unless a client is already waiting. Try running this example with the NP_NOWAIT option instead of NP_WAIT to illustrate the failure.

Figure 11-5 introduces several new APIs. They follow.

DosCreateNPipe

```
         DosCreateNPipe(
                        PSZ   pipename,
                        PHPIPE       pphpipehandle,
                        ULONG        openmode,
                        ULONG        pipemode,
                        ULONG        writebufsize,
                    ULONG        readbufsize,
                    ULONG        timeout);
```

Parameters:

- ❑ pipename (PSZ) input—The name of the pipe. The string has the following syntax: "\\PIPE\\mypipe" where mypipe is whatever you want to name the pipe, within the DOS and OS/2 1.x 8.3 naming convention.

- ❑ pphpipehandle (PHPIPE) output—The pipe will have one handle for each process that attaches to it.

- ❑ openmode (ULONG) input—Set of flags describing attributes of the pipe. The flags are:

Name	**Value**	**Description**
NP_INHERIT	0x0000	Child processes inherit the pipe handle
NP_NOINHERIT	0x0080	Child processes do not inherit the pipe handle
NP_ACCESS_INBOUND	0x0000	Data is written by the client and read by the server
NP_ACCESS_OUTBOUND	0x0001	Data is written by the server and read by the client
NP_ACCESS_DUPLEX	0x0002	Data is written and read by both the server and the client. (However, OS/2 2.1 ensures that neither reads the same data it wrote.)

The next two flags are ignored if the pipe is local:

NP_WRITEBEHIND	0x0000	Allow the remote system to cache data before writing it to the pipe.
NP_NOWRITEBEHIND	0x4000	Do not allow the remote system to cache data before writing it to the pipe.

- pipemode (ULONG) input—Set of flags describing other attributes of the pipe. The flags are:

Name	**Value**	**Description**
NP_WAIT	0x0000	The pipe will block on read or write, if the operation cannot be completed immediately.
NP_NOWAIT	0x8000	The pipe will always return immediately from a read or write operation. If a read cannot complete, it returns ERROR_NO_DATA. If a write cannot complete, it returns with the byteswritten field set to 0.
NP_TYPE_BYTE	0x0000	The pipe is a byte pipe.
NP_TYPE_MESSAGE	0x0400	The pipe is a message pipe.
NP_READMODE_BYTE	0x0000	Read from the pipe as though it were a byte pipe.
NP_READMODE_MESSAGE	0x0100	Read from the pipe as though it were a message pipe. This is invalid if the NP_TYPE_BYTE flag is set.
Bits 7-0	(0x___N)	The number of instances of the pipe that can exist simultaneously. Values are 1-254, or -1 (255). -1 means there is no limit to the number of instances that can exist. This field is ignored if an instance of the pipe already exists. Note that 0 is not a valid value.

- writebufsize (ULONG) input —The size of the buffer for writing messages from the server to the client.

- readbufsize (ULONG) input—The size of the buffer for writing messages from the client to the server.

- timeout (ULONG) input—Default timeout, in milliseconds, for clients using the DosWaitNPipe API. If this field is 0, a system default will be used. If no default is set, the default will be wait-indefinitely.

Note: The client can override this value on the DosWaitNPipe API.

DosConnectNPipe

```
        DosConnectNPipe(
                       HPIPE phpipehandle)
```

Parameter:

- ❑ hpipehandle (HPIPE) input—The handle of the pipe to connect to.

DosWaitNPipe

```
        DosWaitNPipe(
                    PSZ    pipename,
                    ULONG  timeout)
```

Parameters:

- ❑ pipename (PSZ) input—The handle of the pipe to connect to.
- ❑ timeout (ULONG) input—Maximum time to wait for a connection before returning. To use the default timeout specified when the pipe was created, set this value to 0. To wait indefinitely, set this value to -1 (0xFFFFFFFF). The system will wait only if the pipe exists; otherwise, the system will return ERROR_BAD_PIPE.

```
/* Attaching to a named pipe.                                      */

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#include <os2.h>

int main (int argc, void *argv[])
{
   #define READBUFSIZE 80
   #define WRITEBUFSIZE 80
   HPIPE   SPipeHndl;                  /* Server handle for pipe1*/
   UCHAR   PipeName[80];               /* name of the named pipe */
   ULONG   PipeDir;                    /* direction of dataflow  */
   ULONG   PipeMode;         /* Blocking/nonblocking,  Byte/Msg..*/
   ULONG   ReadBuf[READBUFSIZE];       /* input buffer           */
   ULONG   WriteBuf[WRITEBUFSIZE];     /* output buffer          */
   ULONG   WriteBufSize;               /* size of output buffer  */
   ULONG   DefWaitTimeOut;             /* default timeout for    */
   strcpy(PipeName, "\\PIPE\\MYPIPE");          /* Name of pipe      */
   PipeDir = NP_ACCESS_DUPLEX;               /* Full duplex pipe      */
         /* Nonblocking, message pipe, max 4 instances                */
   PipeMode = NP_WAIT | NP_TYPE_BYTE | 0x04;
   DefWaitTimeOut = 2000;                   /* Timeout in 2 seconds   */
       /* i.e. if a client does a DosWaitNPipe without               */
       /* specifying any timeout, it will only wait for              */
       /* 2 seconds before timing out.                               */

   DosCreateNPipe(PipeName,                 /* Create one instance of */
                  &SPipeHndl,               /* the pipe.              */
                  PipeDir,
                  PipeMode,
                  WRITEBUFSIZE,
                  READBUFSIZE,
                  DefWaitTimeOut);

   DosConnectNPipe(SPipeHndl);              /* Connect to server end  */
                                            /* of the pipe, and wait  */
                                            /* for a client to attach */

   /* The next two lines are cleanup. A real program would           */
   /* not disconnect right away.  See Figure 11-6.                   */
   DosDisConnectNPipe(SPipeHndl);
   DosClose(SPipeHndl);

}
```

A. Server actions

```
/*****************************************************************/

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#define INCL_ERRORS
#include <os2.h>

int main (int argc, void *argv[])
{

   HPIPE   CPipeHndl;                       /* Client handle for pipe1*/
   UCHAR   PipeName[80];                    /* name of the named pipe */
   ULONG   DefWaitTimeOut;                  /* default timeout for    */
                                            /*     DosWaitNPipe API   */
   /* The next variables are for the DosOpen call                    */
   ULONG    Action;
   APIRET  rc;

   strcpy(PipeName, "\\PIPE\\MYPIPE");          /* Name of pipe     */
   rc = ERROR_PIPE_BUSY;                        /* Start the loop   */

   while (rc == ERROR_PIPE_BUSY)  {
      rc = DosWaitNPipe(PipeName,
                        -1);                   /* Wait indefinitely */
      if (!rc) {                        /* Pipe instance is available */
```

```
         rc = DosOpen(PipeName,
                 &CPipeHndl,
                 &Action,                        /* output status     */
                 0,                              /* ignored           */
                 0,                              /*                   */
                 FILE_OPEN,
                 OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
                 0);                             /* No extended attrs */
      }
      else  {
         printf("rc from doswait is: %ld\n", (long) rc);
      }                                                 /* endif */
   }
   printf("final rc is: %ld\n", (long) rc);

   DosClose(CPipeHndl);
}
                          B.  Client actions
/**********************************************************************/
```

Figure 11-5. Attaching to a named pipe— server and client actions.

PIPE STATES

A named pipe can be in any of four *states*:

- ❑ Listening
- ❑ Connected
- ❑ Disconnected
- ❑ Closing

Data can be written to or read from a pipe only in the connected state, so it is important to know what state the pipe is in. When the pipe is first created, by means of the DosCreateNPipe API, it is put in the disconnected state. This means that the pipe exists, but neither the server nor a client can access it.

The next thing that must happen is for the server to issue the DosConnectNPipe API. At this point the pipe is put in the listening state. This means that the server is ready to exchange data with a client, but there is no client currently at the other end. When a client issues a DosOpen for the pipe, it will be put in the connected state.

At this point, both the server and the client can read and write to the pipe (depending on the direction of the pipe, of course). If the client does a DosClose while the pipe is connected, the pipe will be put in the closing state. The server should not do a DosClose from the connected state; it should wait for the client to close its end of the pipe first. Then the server can issue a DosDisConnectNPipe, followed by DosClose.

Some of the error codes pertain directly to the state of a pipe. ERROR_BAD_PIPE means that no pipe with that name exists. ERROR_PIPE_BUSY means that the pipe exists, but there is no instance of the pipe currently available in the listening state. ERROR_BROKEN_PIPE means that one end of the pipe has been closed.

Other Ways to Access a Message Pipe

You have seen that a pipe can be read from and written to by using DosRead and DosWrite. When a client and server exchange information in an asynchronous manner, that is, either one sending several messages without waiting to receive responses in between, using DosRead and DosWrite is the only way to do it. More commonly, a client will send a request to the server and wait for a reply. The client may send several requests in this way, waiting for a response each time, or it may send only one. For these more common scenarios, depicted in Figure 11-6, OS/2 2.1 provides a simple protocol.

```
/*  Common usage scenario.                                        */

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#define INCL_ERRORS
#include <os2.h>

int main (int argc, void *argv[])
{
   #define MAX_NAME_LEN   80
   #define REQ_BUF_SIZE   80
   #define REPLY_BUF_SIZE 80
   HPIPE   CPipeHndl;                  /* Client handle for pipe1 */
   UCHAR   PipeName[MAX_NAME_LEN];     /* name of the named pipe  */
   UCHAR   Request[REQ_BUF_SIZE];      /* Buffer Client's request */
   UCHAR   Reply[REPLY_BUF_SIZE];      /* Buffer Server's reply   */
   ULONG   num;
   APIRET  rc;
      /* The next variables are for the DosOpen call              */
   ULONG    Action;

   strcpy(PipeName, "\\PIPE\\MYPIPE");          /* Name of pipe     */
   strcpy(Request, "How much did I spend on catfood last
year?");

   /* This client only sends one request to the server. It          */
   /* will try twice to connect to the server, and give up          */
   /* if still unsuccessful.                                        */

   do {

      rc = DosOpen(PipeName,
            &CPipeHndl,
            &Action,                         /* output status     */
            0,                               /* ignored           */
            0,                               /*                   */
            FILE_OPEN,
            OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
            0);                              /* No extended attrs */

      if (rc == ERROR_PIPE_BUSY) {           /* Try one more time */
            /* Use the default timeout                            */
         rc = DosWaitNPipe(PipeName, 0);
         if ( rc ) break;
         rc = DosOpen(PipeName, &CPipeHndl, &Action, 0, 0,
            FILE_OPEN,
            OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0);
      }                                               /* endif */

      if (rc){
         printf("Could not open the pipe\n");
         break;
      }

      DosWrite(CPipeHndl, Request, 80, &num);      /* Send request */
      DosRead( CPipeHndl, Reply, 80, &num);        /* Get response */

      printf("Reply from the server is:  %s\n", Reply);

      DosClose(CPipeHndl);

   } while (0 );              /* enddo - fall out of loop          */

}

/******************************************************************/
/* Common usage scenario 1 - client side                          */
/******************************************************************/

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#define INCL_DOSERRORS
#include <os2.h>
```

```
int main (int argc, void *argv[])
{

   #define MAX_NAME_LEN   80
   #define REQ_BUF_SIZE   80
   #define REPLY_BUF_SIZE 80
   HPIPE   CPipeHndl;                    /* Client handle for pipe */
   UCHAR   PipeName[MAX_NAME_LEN];       /* name of the named pipe */
   UCHAR   Request[REQ_BUF_SIZE];        /* Buffer Client's request*/
   UCHAR   Reply[REPLY_BUF_SIZE];        /* Buffer Server's reply  */
   ULONG   num;
   APIRET  rc;

   /*      The next variables are for the DosOpen call             */
   ULONG    Action;

   strcpy(PipeName, "\\PIPE\\MYPIPE");          /* Name of pipe    */
   strcpy(Request, "How much did I spend at the vet last year?");

   /*   This client sends several requests to the server           */

   do {

     rc = DosOpen(PipeName,
            &CPipeHndl,
            &Action,                          /* output status     */
            0,                                /* ignored           */
            0,                                /*                   */
            FILE_OPEN,
            OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
            0);                               /* No extended attrs */
     if (rc == ERROR_PIPE_BUSY) {             /* Try one more time */
        /* Use the default timeout                                 */
        rc = DosWaitNPipe(PipeName, 0);
        if ( rc ) break;
        rc = DosOpen(PipeName, &CPipeHndl, &Action, 0, 0,
             FILE_OPEN,
             OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0);
     }                                                 /* endif */

     if (rc)  {
        printf("Could not open the pipe\n");
        break;
     }

     DosWrite(CPipeHndl, Request, 80, &num);        /* Send request */
     DosRead( CPipeHndl, Reply, 80, &num);          /* Get response */
     printf("First response was:  %s\n", Reply);

     strcpy(Request, "Transfer money to checking account\n");
     DosWrite(CPipeHndl, Request, 80, &num);        /* Send request */
     DosRead( CPipeHndl, Reply, 80, &num);          /* Get response */
     printf("Second response was:  %s\n", Reply);
     strcpy(Request, "Get paperwork for a loan\n");
     DosWrite(CPipeHndl, Request, 80, &num);        /* Send request */
     DosRead( CPipeHndl, Reply, 80, &num);          /* Get response */
     printf("Third response was:  %s\n", Reply);

     DosClose(CPipeHndl);

   } while (0 );                             /* enddo - fall out of loop */
}                                            /* End main                 */

/*******************************************************************/
/* Common usage scenario 2 - client side                          */
/*******************************************************************/

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#include <os2.h>

int main (int argc, void *argv[])
{

   HPIPE   SPipeHndl;                      /* Server handle for pipe1*/
   UCHAR   PipeName[80];                   /* name of the named pipe */
   ULONG   PipeDir;                       /* direction of dataflow   */
   ULONG   PipeMode;              /* Blocking/nonblocking,  Byte/Msg..*/
   ULONG   ReadBufSize;                    /* size of input buffer   */
   ULONG   WriteBufSize;                   /* size of output buffer  */
   ULONG   DefWaitTimeOut;            /*default timeout of DosWaitNPipe*/
   UCHAR   Request[80];
   UCHAR   Reply[80];
              /* Loop through a given set of responses                */
   PSZ     Myrep[5]= {"1", "2", "3", "4", "5"};
   ULONG   len;
   ULONG   num;
   ULONG   i;
   ULONG   j;
   APIRET  rc;

   /* For initializations, see Figure 11-5 */
   strcpy(PipeName, "\\PIPE\\MYPIPE");       /* Name of pipe         */
   PipeDir = NP_ACCESS_DUPLEX;               /* Full duplex pipe     */
              /* Nonblocking, message pipe, max 4 instances          */
   PipeMode = NP_WAIT | NP_WMESG | NP_RMESG | 0x04;
   ReadBufSize = 4096;
   WriteBufSize = 4096;
   DefWaitTimeOut = 2000;               /* Timeout in 2 seconds      */

   rc=DosCreateNPipe(PipeName,          /* Create one instance of    */
                 &SPipeHndl,            /* the pipe.                 */
                 PipeDir,
                 PipeMode,
                 WriteBufSize,
                 ReadBufSize,
                 DefWaitTimeOut);
```

```
    j = 5;
    for   (i=0; i<5; i++) {                    /* Connect up to 5 times    */
    rc=DosConnectNPipe(SPipeHndl);             /* Connect to server end    */
                                               /* of the pipe, and wait    */
                                               /* for a client to attach   */

    rc= DosRead( SPipeHndl, Request, 80, &num);          /* Get request */

    while (num != 0)                       /* Client sent another request */
    {
       printf("The request is:  %s\n", Request);
       if (j == 5) j=0;                /* Only 5 hard-coded responses   */
       strcpy(Reply, Myrep[j++]);                     /* handle it     */
       len = strlen(Reply) + 1;
       rc=DosWrite(SPipeHndl, Reply, len, &num);      /* Send reply    */
       rc=DosRead( SPipeHndl, Request, 80, &num);     /* Get request   */
    }                                                 /* endwhile      */

    rc=DosDisConnectNPipe(SPipeHndl);

    rc=DosClose(SPipeHndl);
    };                                              /* end outer loop */
}
```

Figure 11-6. *Common usage scenario—server side (same for both scenarios).*

DosTransactNPipe combines DosWrite and DosRead in a single call. This can be used by the server but is more commonly used when a client wants to make several synchronous requests. The pipe must be a duplex message pipe, or the return code will be ERROR_BAD_FORMAT. The client must explicitly open and close the pipe outside of this call.

Figure 11-7 shows the use of DosTransactNPipe. Compare it to Figure 11-6 to contrast DosTransactNPipe with DosRead and DosWrite. Using DosTransactNPipe makes the code easier to write, read, and maintain.

Figure 11-7 introduces the DosTransactNPipe API:

DosTransactNPipe

```
          DosTransactNPipe(
                             HPIPE      hpipehandle,
                             PVOID      pWriteBuffer,
                             ULONG      ulSizeofWriteBuffer,
                             ULONG      pReadBuffer,
                             ULONG      ulSizeofReadBuffer,
                             PULONG     pBytesRead);
```

Parameters:

- ❑ hpipehandle (HPIPE) input—The handle of the pipe to connect to.
- ❑ pWriteBuffer (PVOID) input—Pointer to data to be written to the pipe.
- ❑ ulSizeofWriteBuffer (ULONG) input—Size of the data to be written (the request).
- ❑ pReadBuffer (PVOID) input—Pointer to where to put the data read from the pipe (the response).
- ❑ ulSizeofReadBuffer (ULONG) input—Size of the buffer for the the data read from the pipe (the response). If there is more data in the pipe, the return code will be ERROR_MORE_DATA.
- ❑ pBytesRead (PULONG) output—Amount of data returned.

```
/*  DosTransactNPipe.                                                */

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#define INCL_ERRORS
#include <os2.h>

int main (int argc, void *argv[])
{

   #define MAX_NAME_LEN    80
   #define REQ_BUF_SIZE    80
   #define REPLY_BUF_SIZE  80
   HPIPE    CPipeHndl;                   /* Client handle for pipe */
   UCHAR    PipeName[MAX_NAME_LEN];      /* name of the named pipe */
   UCHAR    Request[REQ_BUF_SIZE];       /* Buffer Client's request*/
   UCHAR    Reply[REPLY_BUF_SIZE];       /* Buffer Server's reply  */
   ULONG    len;
   ULONG    num;
   APIRET   rc;

   /* The next variables are for the DosOpen call                    */
   ULONG     Action;

   strcpy(PipeName, "\\PIPE\\MYPIPE");          /* Name of pipe    */
   strcpy(Request, "How much did I spend at the vet last year?");

   /* This client sends several requests to the server               */
```

```
do {

   rc = DosOpen(PipeName,
         &CPipeHndl,
         &Action,                          /* output status     */
         0,                                /* ignored           */
         0,                                /*                   */
         FILE_OPEN,
         OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
         0);                               /* No extended attrs */
   if (rc == ERROR_PIPE_BUSY){             /* Try one more time */
      /* Use the default timeout                                */
      rc = DosWaitNPipe(PipeName, 0);
      if ( rc ) break;
      rc = DosOpen(PipeName, &CPipeHndl, &Action, 0, 0,
         FILE_OPEN,
         OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0);
   }                                                /* endif */

   if (rc) {
      printf("Could not open the pipe\n");
      break;
   }

   len = strlen(Request) + 1;
   DosTransactNPipe(CPipeHndl,
                  Request,               /* Client's request   */
                  len,                   /* len of request     */
                  Reply,                 /* where to put resp  */
                  REPLY_BUF_SIZE,        /* max length of resp */
                  &num                   /* actual len of resp */
                 );

   printf("First response was:  %s\n", Reply);

   strcpy(Request, "Transfer money to checking account\n");
   len = strlen(Request) + 1;
   DosTransactNPipe(CPipeHndl,
                  Request,               /* Client's request    */
                  len,                   /* len of request      */
                  Reply,                 /* where to put resp   */
                  REPLY_BUF_SIZE,        /* max length of resp  */
                  &num                   /* actual len of resp  */
                 );
   printf("Second response was:  %s\n", Reply);

   strcpy(Request, "Get paperwork for a loan\n");
   len = strlen(Request) + 1;
   DosTransactNPipe(CPipeHndl,
                  Request,               /* Client's request    */
                  len,                   /* len of request      */
                  Reply,                 /* where to put resp   */
                  REPLY_BUF_SIZE,        /* max length of resp  */
                  &num                   /* actual len of resp  */
```

```
                     );
      printf("Third response was:  %s\n", Reply);

      DosClose(CPipeHndl);
    } while (0 );                        /* enddo - fall out of loop */
}
```

Figure 11-7. *Use of DosTransactNPipe.*

DosCallNPipe goes one step further, combining DosOpen, DosWrite, DosRead, and DosClose in a single call. This is very useful when a client wants to send only one request. The pipe must be a duplex message pipe, or the return code will be ERROR_BAD_FORMAT. If the pipe is busy, that is, there is a client attached to every instance of the pipe, then this call will block for as long as you want. Figure 11-8 shows the the use of DosCallNPipe. Compare it to the single request scenario in Figure 11-6.

Figure 11-8 introduces the DosCallNPipe API:

DosCallNPipe

```
DosCallNPipe(
              PSZ         pszPipeName,
              PVOID       pWriteBuffer,
              ULONG       ulSizeofWriteBuffer,
              ULONG       pReadBuffer,
              ULONG       ulSizeofReadBuffer,
              PULONG      pBytesRead,
              ULONG       ulTimeout);
```

Parameters:

- ❑ pszPipeName (PSZ) input—The name of the pipe to connect to.
- ❑ pWriteBuffer (PVOID) input—Pointer to data to be written to the pipe.
- ❑ ulSizeofWriteBuffer (ULONG) input—Size of the data to be written (the request).
- ❑ pReadBuffer (PVOID) input—Pointer to where to put the data read from the pipe (the response).

- ❑ ulSizeofReadBuffer (ULONG) input—Size of the buffer for the the data read from the pipe (the response). If there is more data in the pipe, the return code will be ERROR_MORE_DATA.

- ❑ pBytesRead (PULONG) output—Amount of data returned.

- ❑ ulTimeOut (ULONG) input—How long to wait for an instance of the pipe to become available. This is the same as the timeout field of the DosWaitNPipe API, introduced in Figure 11-5.

```
/*  DosCallNPipe.                                                  */

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#define INCL_ERRORS
#include <os2.h>

int main (int argc, void *argv[])
{
   #define MAX_NAME_LEN   80
   #define REQ_BUF_SIZE   80
   #define REPLY_BUF_SIZE 80
   UCHAR    PipeName[MAX_NAME_LEN];        /* name of the named pipe */
   UCHAR    Request[REQ_BUF_SIZE];         /* Buffer Client's request*/
   UCHAR    Reply[REPLY_BUF_SIZE];         /* Buffer Server's reply  */
   ULONG    num;
   ULONG    len;
   APIRET   rc;

   strcpy(PipeName, "\\PIPE\\MYPIPE");          /* Name of pipe      */
   strcpy(Request, "How much did I spend at the vet last year?");
   len = strlen(Request) + 1;
   /* This client sends one requests to the server                   */
   rc = DosCallNPipe( PipeName,
                   Request,                 /* Client's request      */
                   len,                     /* len of request        */
                   Reply,                   /* where to put response */
                   REPLY_BUF_SIZE,          /* max length of response*/
                   &num,                    /* actual len of response*/
                   2000                     /* timeout in 2 seconds  */
                  );
   if (!rc) printf("Response was:  %s\n", Reply);
   else     printf("DosCallNPIpe failed\n");
}
```

Figure 11-8. *Use of DosCallNPipe.*

A main difference between the DosTransactNPipe and the DosCallNPipe API is that DosTransactNPipe allows multiple synchronous requests without opening and closing the pipe each time. If you were to use DosCallNPipe in Figure 11-7, the pipe would be implicitly opened and closed three times. Performance will be better if the pipe is only opened and closed once.

OBTAINING AND CHANGING INFORMATION ABOUT A NAMED PIPE

Since a named pipe is so flexible, there ought to be a way to determine the characteristics of a particular pipe, and there is. DosQueryNPHState returns information about both the pipe as a whole and the end of the pipe attached to the caller, as follows:

- ❑ Whether the pipe is a byte pipe or a message pipe
- ❑ How many instances of the pipe are allowed
- ❑ Whether that end of the pipe is blocking or nonblocking
- ❑ Whether that end of the pipe is the client or the server
- ❑ Whether that end of the pipe is reading in byte mode or in message mode (a byte pipe can be read only in byte mode)

The pipe type and instance count are fixed when the pipe is created, and the server is always the process that issued the DosCreateNPipe. The blocking mode and reading mode, however, are changeable, by calling DosSetNPHState. When a client initially opens the pipe, the pipe is automatically set to blocking and byte-reading mode. Figure 11-9 shows how a client would change the reading mode of a pipe, and Figure 11-10 shows how one could code a generic routine to terminate the use of a pipe.

DosQueryNPipeInfo returns other fixed information about a pipe, as follows:

- ❑ The actual size of the inbound and outbound pipe buffers—When a pipe is created, you specify a size for the pipe. The system then allocates one or two buffers (two if it is a duplex pipe) for data. If there is not enough memory to allocate the size you specified, then the system will allocate a smaller size.

- ❑ The maximum number of instances allowed for the pipe and the number of instances currently existing—The first time DosCreateNPipe is called with a particular name, an instance of the pipe is created, and the maximum number of instances allowed for pipes with that name is set to the number given in the call. Subsequent DosCreateNPipe calls with the same name will cause another instance to be created, up to the maximum number specified on the first call. The maximum number field in this call can be ignored. An instance ceases to exist when both ends are closed.

- ❑ The name of the pipe and the length of the name—If the pipe is remote, the name will include the HostName, that is, the name of the system on which the pipe was created.

- ❑ A unique 2-byte identifier for the client—This information can be requested instead of all the other information. It is used by servers when multiple instances exist, and the server does not dedicate a thread to each instance. This is discussed further in the next section.

Figure 11-9 introduces the DosSetNPHState API.

DosSetNPHState

```
DosSetNPHState(
                HPIPE    hpipehandle,
                ULONG    pipestate);
```

Parameters:

- ❑ hpipehandle (HPIPE) input—The handle of the pipe whose state you want to change.

- ❑ pipestate (ULONG) input—Bit data about the new state. You can change the blocking mode of the pipe and the read mode of the pipe:

Name	**Value**	**Description**
NP_WAIT	0x0000	The pipe will block on read or write, if the operation cannot be completed immediately.
NP_NOWAIT	0x8000	The pipe will always return immediately from a read or write operation. If a read cannot complete, it returns ERROR_NO_DATA. If a write cannot complete, it returns with the byteswritten field set to 0.
NP_READMODE_BYTE	0x0000	Read from the pipe as though it were a byte pipe.
NP_READMODE_MESSAGE	0x0100	Read from the pipe as though it were a message pipe. This is invalid if the NP_TYPE_BYTE flag is set.

The counterpart to DosSetNPHState is DosQueryNPHState.

DosQueryNPHState

```
        DosQueryNPHState(
                        HPIPE    hpipehandle,
                        PULONG   ppipestate);
```

Parameters:

- ❑ hpipehandle (HPIPE) input—The handle of the pipe whose state you want to know.

- ❑ pipestate (PULONG) output—Bit data about the state. This will tell you about five attributes:

  Blocking Mode:

NP_WAIT	0x0000	These flags are the same as
NP_NOWAIT	0x8000	on the DosSetNPHState API.

  Byte or Message pipe:

NP_TYPE_BYTE	0x0000	This is a byte pipe.
NP_TYPE_MESSAGE	0x0400	This is a message pipe.

Whether the pipe is being read as bytes or messages. These flags are the same as on the DosSetNPHState API:

```
NP_READMODE_MESSAGE   0x0100  Read as messages
NP_READMODE_BYTE      0x0100  Read as bytes
```

Whether the handle belongs to the server or a client:

```
NP_END_CLIENT         0x0000  Belongs to the client
NP_END_SERVER         0x4000  Belongs to the server
```

How many instances of the pipe can be created:

```
Bits 7—0.
```

```
/*  DosSetNPHState                                                    */

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#define INCL_ERRORS
#include <os2.h>

int main (int argc, void *argv[])
{
   #define REQ_BUF_SIZE   80
   #define REPLY_BUF_SIZE 80

   HPIPE   CPipeHndl;                     /* Client handle for pipe */
   UCHAR   PipeName[80];                  /* name of the named pipe */
   UCHAR   Request[REQ_BUF_SIZE];         /* Buffer Client's request*/
   UCHAR   Reply[REPLY_BUF_SIZE];         /* Buffer Server's reply  */
   ULONG   len;
   ULONG   num;
   APIRET  rc;

   /*      The next variables are for the DosOpen call                */
   ULONG    Action;

   strcpy(PipeName, "\\PIPE\\MYPIPE");            /* Name of pipe   */
   strcpy(Request, "How much did I spend at the vet last year?");
   len = strlen(Request) + 1;

   do {

     /* Open the pipe                                                 */
     rc = DosOpen(PipeName,
             &CPipeHndl,
             &Action,                          /* output status     */
             0,                                /* ignored           */
             0,                                /*                   */
```

```
                FILE_OPEN,
                OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
                0);                              /* No extended attrs */

        if (rc) break;
        DosWrite(CPipeHndl, Request, len, &num);       /* Send request */
        DosSleep(1000);         /* Give the server time to respond    */
                                /* If the client has higher priority, */
                                /* then DosRead should get an error   */

        /*Set the pipe to read in message mode, and to not block       */
        rc = DosSetNPHState(CPipeHndl, NP_RMESG | NP_NOWAIT);

        rc = DosRead( CPipeHndl, Reply, 80, &num);       /*Get response*/
        if (!rc) printf("Reply was:  %s\n", Reply);
        else     printf("Did not have any data to read\n");
        DosClose(CPipeHndl);
     } while (0 );                                         /* enddo */
}
```

Figure 11-9. *Use of DosSetNPHState.*

Figure 11-10 introduces the DosPeekNPipe API, which returns information about data in the pipe, without removing any data from the pipe:

DosPeekNPipe

```
          DosPeekNPipe(
                       HPIPE        hpipehandle,
                       PVOID        buffer,
                       ULONG        bufferlen,
                       PULONG       bytesread,
                       PAVAILDATA   availdata,
                       PULONG       ppipestate);
```

Parameters:

- ❑ pipehandle (HPIPE) input—The handle of the pipe to peek into.
- ❑ buffer (PVOID) input/output —Where to place data from the pipe

  Note: The data remains in the pipe as well.
- ❑ bufferlen (ULONG) input—Length of the buffer.
- ❑ bytesread (ULONG) output—How many bytes were read.

- availdata (PAVAILDATA) output—Structure containing two values:

  Bits 32-16: The number of bytes of data in the pipe, including message headers if it is a message pipe.

  Bits 15-0: For a mesage pipe, the number of bytes in the next message to be read; for a byte pipe, 0.

- pipestate (ULONG) output—The *state* of the pipe:

NP_STATE_DISCONNECTED	0x0001
NP_STATE_LISTENING	0x0002
NP_STATE_CONNECTED	0x0003
NP_STATE_CLOSING	0x0004

  (These states are described in detail earlier in this chapter in *Pipe States.)*

```
/*   Closing a pipe.                                                    */

#define INCL_DOSNMPIPES
#define INCL_DOSFILEMGR
#include <os2.h>

closepipe(pipehndl)
{
   ULONG    PipeStateInfo;
   ULONG    bytesread;
   struct _AVAILDATA bytesavail;
   ULONG    pipestate;
   ULONG    pipeinfo;
   APIRET   rc;

   do {
      /* Determine if the pipe is in use, disconnected,                 */
      /* or already closed                                              */
      rc = DosPeekNPipe(pipehndl,
                        NULL,                        /* ptr to buffer */
                        0,                           /* len of buffer */
                        &bytesread,
                        &bytesavail,
                        &pipestate);
      switch(rc) {
         case NO_ERROR:
            if (pipestate == NP_STATE_CONNECTED) {
               /* If this is the client, close it                       */
               /* If this is the server, wait for                       */
               /*    the client to close it first                       */
               rc = DosQueryNPHState(pipehndl, &pipeinfo);
               if (!rc && (pipeinfo & NP_END_SERVER))
                  rc = 1;                       /* this is a server */
               else if (!rc)
                  DosClose(pipehndl);           /* this is a client */
            }
            else                            /* pipe is not connected */
               DosClose(pipehndl);
            break;
         case ERROR_BAD_PIPE:
            rc = 0;                         /* exit the loop          */
            break;                          /* pipe is already closed */
         case ERROR_PIPE_BUSY:
            /* Should use a semaphore, but since these                */
            /* have not been introduced yet, use sleep                */
            DosSleep(20000);                /* Try every 2 seconds    */
            break;
         case ERROR_PIPE_NOT_CONNECTED:
            DosClose(pipehndl);
            rc = 0;                         /* to exit the loop       */
            break;

      }                                               /* endswitch */

   }   while (rc != NO_ERROR);                           /* enddo */
}
```

Figure 11-10. A generic routine to close a pipe.

COORDINATING MULTIPLE INSTANCES WITH SEMAPHORES

In a single-server, multiple-client environment, the server might want to create several instances of a pipe so that it can serve several clients at the same time. Each instance of the pipe must be created by calling DosCreateNPipe, and each instance will have its own handle.

Figure 11-11 shows a server creating two instances of a pipe.

```
/*  Two named pipes.                                                  */

#define INCL_DOSNMPIPES
#define INCL_ERRORS
#include <os2.h>
int main (int argc, void *argv[])
{
   #define READBUFSIZE  400
   #define WRITEBUFSIZE  400
   HPIPE   SPipeHndl1;         /* Server handle for pipe, instance 1*/
   HPIPE   SPipeHndl2;         /* Server handle for pipe, instance 2*/
   UCHAR   PipeName[80];                /* name of the named pipe */
   ULONG   PipeDir;                     /* direction of dataflow  */
   ULONG   PipeMode;            /* Blocking/nonblocking,  Byte/Msg..*/
   ULONG   ReadBuf[READBUFSIZE];        /* input buffer           */
   ULONG   WriteBufSize[WRITEBUFSIZE];  /* output buffer          */
   ULONG   DefWaitTimeOut;              /* default timeout for    */
                                        /*     DosWaitNPipe API   */
   APIRET  rc;

   strcpy(PipeName, "\\PIPE\\MYPIPE");  /* Name of pipe           */
   printf("pipename is %s\n", PipeName);
   PipeDir = NP_ACCESS_DUPLEX;          /* Full duplex pipe       */
            /* Nonblocking, message pipe, max 4 instances         */
   PipeMode = NP_NOWAIT | NP_WMESG | NP_RMESG | 0x04;
   DefWaitTimeOut = 2000;              /* Timeout in 2 seconds    */
       /* i.e. if a client does a DosWaitNPipe without            */
       /* specifying any timeout, it will only wait for           */
       /* 2 seconds before timing out.                            */

   DosCreateNPipe(PipeName,            /* Create one instance of  */
                  &SPipeHndl1,         /* the pipe.               */
                  PipeDir,
                  PipeMode,
                  WRITEBUFSIZE,
                  READBUFSIZE,
                  DefWaitTimeOut);
   DosCreateNPipe(PipeName,            /* Create second instance  */
                  &SPipeHndl2,         /* of the pipe.            */
                  PipeDir,
                  PipeMode,
                  WRITEBUFSIZE,
                  READBUFSIZE,
                  DefWaitTimeOut);

   DosConnectNPipe(SPipeHndl1);           /* Connect to instances */
   DosConnectNPipe(SPipeHndl2);
}
```

Figure 11-11. *Creating two instances of a named pipe.*

Now the server is going to wait for a client to attach to one or both of the pipes and for a request to come through. You could dedicate a thread to each instance, or you could poll each pipe periodically. If the pipe is local, you could use a semaphore.

There are two different ways of connecting semaphores with named pipes: *shared event* semaphores or *muxwait* semaphores. To use shared event semaphores, you create a shared event semaphore and attach it to one or more pipes by calling DosSetNPipeSem. Then you will wait on that semaphore and query it (by using DosQueryNPipeSemState) to find out which pipe had activity (see Figure 11-12).

To use muxwait semaphores, you create an event semaphore for each pipe, attach them to the pipes, create a muxwait semaphore to wait on any one to be active, and attach it to the event semaphores. Then you wait on the muxwait semaphore and query it to find out which pipe had activity. (In this discussion, each instance of a pipe is considered a separate pipe.)

Figure 11-12 shows a server attaching an event semaphore to each of the two instances of the pipe created in Figure 11-11. The key handle is assigned by the process attaching the semaphore and is the value returned by DosQueryNPipeInfo for Level 2 information. A pipe can have a semaphore attached to each end (but only one because attaching a second detaches the first). In this case, each end must specify a key handle. The key handle does not have to be the same at both ends.

```
/*********************************************************************/
/* This figure is a continuation of Figure 11-11.  Assume           */
/* that SPipeHndl1 and SPipeHndl2 are the pipe handles              */
/* from Fig. 11-11, and that both instances are connected.          */
/* Assume the following routines exist:                             */
/*    GetUniqueID - returns a different number each time            */
/*    ProcessPipe - processes data in a pipe, given its             */
/*                        handle                                    */
/*    closepipe   - from Figure 11-10                               */


HEV    SemHndl;
HPIPE  currentpipe;
ULONG  KeyValue1;               /* System-wide unique identifier   */
ULONG  KeyValue2;               /* System-wide unique identifier   */
CHAR   infobuf[100];            /* Buffer for data about semaphore */
struct {
   BYTE   status;
   BYTE   flag;
   ULONG  keyvalue;
   USHORT num;
} *RecordPtr;
LONG pipesactive = 2;
   /* Create a shared, unnamed event semaphore                      */
rc = DosCreateEventSem (NULL,                 /* unnamed         */
                        &SemHndl,             /* ptr to handle   */
                        DC_SEM_SHARED,        /* shared          */
                        0);                   /* initially 'set' */
KeyValue1 = GetUniqueID();
KeyValue2 = GetUniqueID();
```

```
rc = DosSetNPipeSem(PipeHndl1,                /* Attach the first    */
                    SemHndl,                  /* instance, with an   */
                    KeyValue1);               /* identifier          */
rc = DosSetNPipeSem(PipeHndl2,                /* Attach the second   */
                    SemHndl,                  /* instance, with a    */
                    KeyValue2);               /* different identifier*/

/* Wait for a client to put data into either pipe                    */
rc = DosWaitEventSem(SemHndl, SEM_INDEFINITE_WAIT);

/* Determine which pipe has data, and act on it                      */
rc = DosQueryNPipeSemState(SemHndl,              /* semaphore handle */
                           &infobuf,             /* where to put data*/
                           100);                 /* length of buffer */

/* There should be two records in the buffer, one for each           */
/* instance of the pipe.                                             */

while(pipesactive != 0){
  RecordPtr = infobuf;
  while (RecordPtr->status != 0){                /* check every record*/

    /* Determine which pipe this record refers to                    */
    if (RecordPtr->keyvalue == KeyValue1)
        currentpipe = PipeHndl1;
    else currentpipe = PipeHndl2;

    if (RecordPtr->status & NPSS_RDATA)           /* data available */
       ProcessPipe(currentpipe);
    if (RecordPtr->status & NPSS_CLOSE){          /* pipe was closed*/
       closepipe(currentpipe);
       pipesactive--;
    }

    RecordPtr++;                     /* look at next record in buffer */
  }                                  /* endwhile                      */
}                                    /* endwhile records in buffer    */
```

Figure 11-12. *Using an event semaphore with pipes*

MORE ON PIPE SYNCHRONIZATION

There are two functions available that haven't been discussed yet: DosResetBuffer and DosPeekNPipe. When used with a file, DosResetBuffer forces the data in the buffer to be written to disk, and the calling process is blocked until that happens.

When used with a pipe, DosResetBuffer blocks the calling process until any data has been read by the process at the other end of the pipe. This happens regardless of the blocking mode at either end of the pipe.

DosPeekNPipe can be used in several places. This function never blocks, and it returns information about data in the pipe and the state of the pipe without actually removing data from the pipe.

DosPeekNPipe returns the number of bytes in the pipe and the number of bytes in the current message. This allows a process to allocate a buffer just large enough to hold the next message, if need be. The number of bytes in the current message is always set to 0 for a byte pipe.

DosPeekNPipe also returns all the data in the pipe, up to the number specified in the call. Since it does not actually change the contents of the pipe, those bytes must still be read. Also, message boundaries are ignored, so the data returned to you may end with only a partial message. This can happen in two ways.

> First, if there is more data in the the pipe than you ask to look at, and the cutoff point is in the middle of a message, you'll get a partial message.
>
> Second, if a process is currently writing into the pipe when you call DosPeekNPipe, only part of the message will be accessible. Since DosPeekNPipe does not block, it will return whatever is accessible in the pipe at the time of your call. The message headers added by the system are skipped over.

DosPeekNPipe also returns the state of the pipe. This corresponds to the four states, disconnected, listening, connected, and closing. This call can be used to determine when a client has opened or closed its end of the pipe.

SUMMARY

This chapter discusses pipes, a powerful form of interprocess communication. It explaines and illustrates the types of pipes available as well as almost all of the pipe-related APIs. The first part of the chapter deals with unnamed pipes. Unnamed pipes can only be used within a process or between a parent process and its children.

They are always duplex. Each unnamed pipe has two handles, one for reading and one for writing.

The second part of the chapter deast with named pipes. Named pipes can be used by any process or thread that knows the pipe's name, even if the process resides on a different machine (if there is a communications link between the machines). Each process or thread that attaches to a named pipe gets two handles, one for reading and one for writing. Threads within a single process can share the same handles. Named pipes can be one way in either direction or they can be duplex. They can be used to transmit data as either a sequence of bytes (a byte pipe) or as logical chunks (a message pipe). Logically, many clients can attach to the same pipe; physically this is accomplished by creating multiple instances of the pipe.

Pipes can be combined with semaphores to provide the most powerful and flexible form of interprocess communication within a system.

The API associated with unnamed pipes is:

- ❑ DosCreatePipe—Creates an unnamed pipe. Then use DosRead, DosWrite, and DosClose.

The APIs associated with named pipes are:

- ❑ DosCallNPipe—Executes a series of commands on a named pipe, namely DosOpen, DosWrite, DosRead, and DosClose; used by the client only.
- ❑ DosConnectNPipe—Puts the pipe into the listening state. Used by the server to indicate it readiness; a client can then do a DosOpen on the pipe to establish a link; used by the server only.
- ❑ DosCreateNPipe—Creates a named pipe. The state of the pipe is disconnected. Used by the server only (the caller becomes the server).
- ❑ DosDisconnectNPipe—Disconnects the server from a named pipe and puts the pipe into the disconnect state; should be done after the client closes its end. All data remaining in the pipe, if any, is lost; used by the server only.
- ❑ DosPeekNPipe—Returns data in the pipe without removing it from the pipe.

- ❑ DosQueryNPHState—Returns information about the pipe handle (whether the corresponding pipe is a byte pipe or a message pipe, whether the current read mode is byte or message, whether the current blocking mode is blocking or nonblocking, whether the handle is owned by the client or the server, and how many instances of the corresponding pipe are allowed).
- ❑ DosQueryNPipeInfo—Returns information about the pipe, namely, the name of the pipe and the length of that name, the size of the pipe's inbound and outbound buffers, how many instances of the pipe are allowed, and the 2–byte identifier for the client end of the pipe.
- ❑ DosQueryNPipeSemState—Returns information about the pipes connected to a semaphore.
- ❑ DosSetNPipeSem—Attaches a shared event semaphore to a named pipe.
- ❑ DosTransactNPipe—Executes a series of commands on a named pipe, namely DosWrite and DosRead.
- ❑ DosWaitNPipe—Waits a specified amount of time for a named pipe to become available. Used by the client only.

Other APIs that can be used with named pipes are:

- ❑ DosOpen
- ❑ DosRead
- ❑ DosWrite
- ❑ DosClose
- ❑ DosResetBuffer

CHAPTER 12

Queues

"I'm making a list." —*Mom*

INTRODUCTION

Queues are the most common method used to communicate information between the threads and the processes of a program. Although most methods of interprocess communication are used to varying degrees in a typical large program, queues will usually be implemented to drive the events that happen during program execution.

In an OS/2 Presentation Manager program, when the user operates an input device such as a mouse, messages are sent through a user input queue to the application. Although OS/2 Presentation Manager (PM) is not a focus of this book, the PM API set used for managing the user input queue is similar in functionality to the one discussed in this chapter.

This chapter will discuss and give examples of the available options that OS/2 2.1 provides in its underlying queue support. This complete set of options includes element ordering, methods for reading and writing elements (elements are objects in the queue), element data structure, and managing multiple queues.

ORDER OF QUEUE ELEMENTS

Although any element can be read and removed off the queue, OS/2 queues provide three ordering mechanisms to help facilitate common queuing data structures. These common data structures are the *stack*, the *queue*, and the *priority queue*.

The stack data structure is one in which each element that is to be added to the stack is added at the front of the stack and each element removed is removed from the front of the stack. This constitutes a Last-In-First-Out (LIFO) ordering. The following figures illustrate a stack (Figure 12-1), adding to a stack (Figure 12-2), and removing an item from a stack (Figure 12-3).

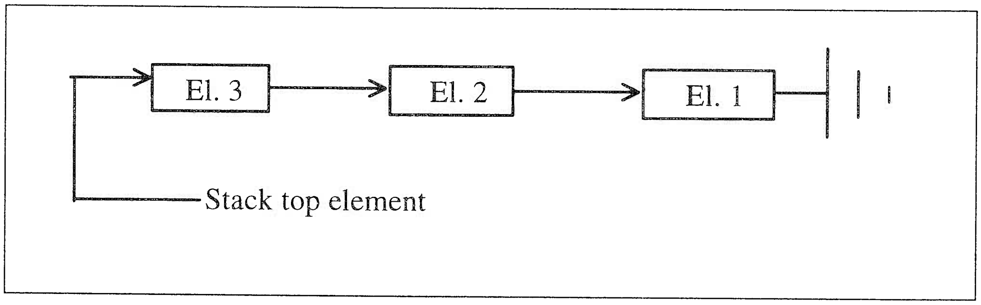


Figure 12-1. Stack illustration.

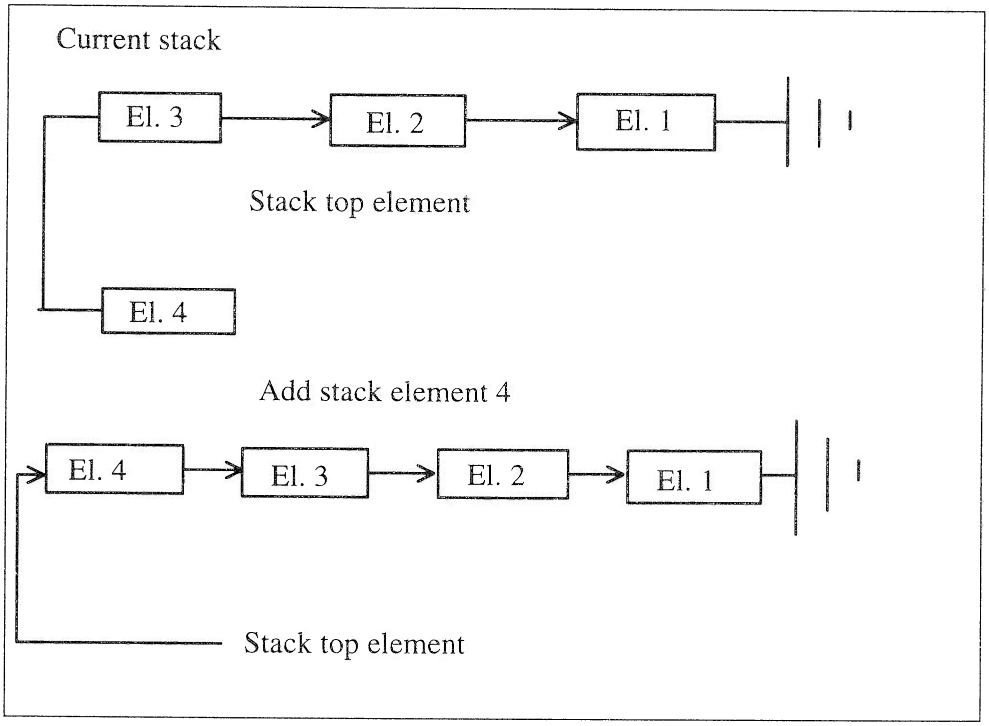


Figure 12-2. Adding to a stack.

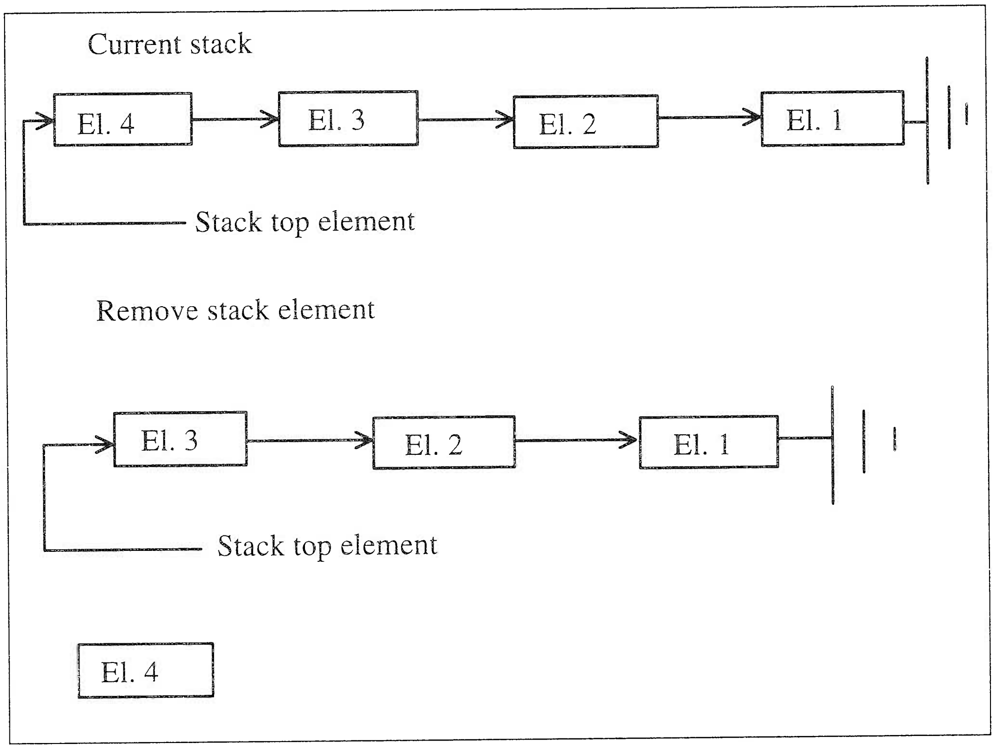


Figure 12-3. *Removing an item from a stack.*

The queue data structure is one in which each element that is to be added to the queue is added at the end of the queue, and each element removed is removed from the front of the queue. This constitutes a First-In-First-Out (FIFO) ordering. The following figures illustrate a queue (Figure 12-4), adding to a queue (Figure 12-5), and removing an item from a queue (Figure 12-6).

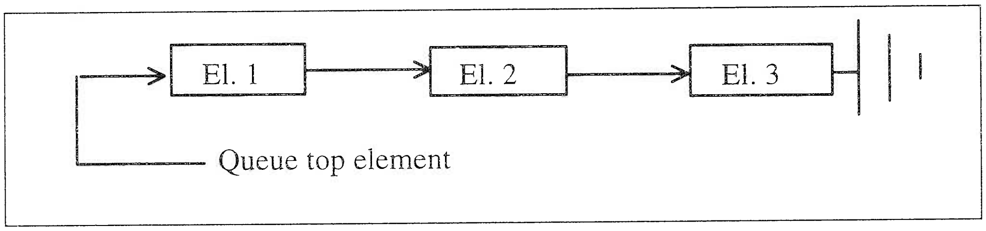


Figure 12-4. *Queue illustration.*

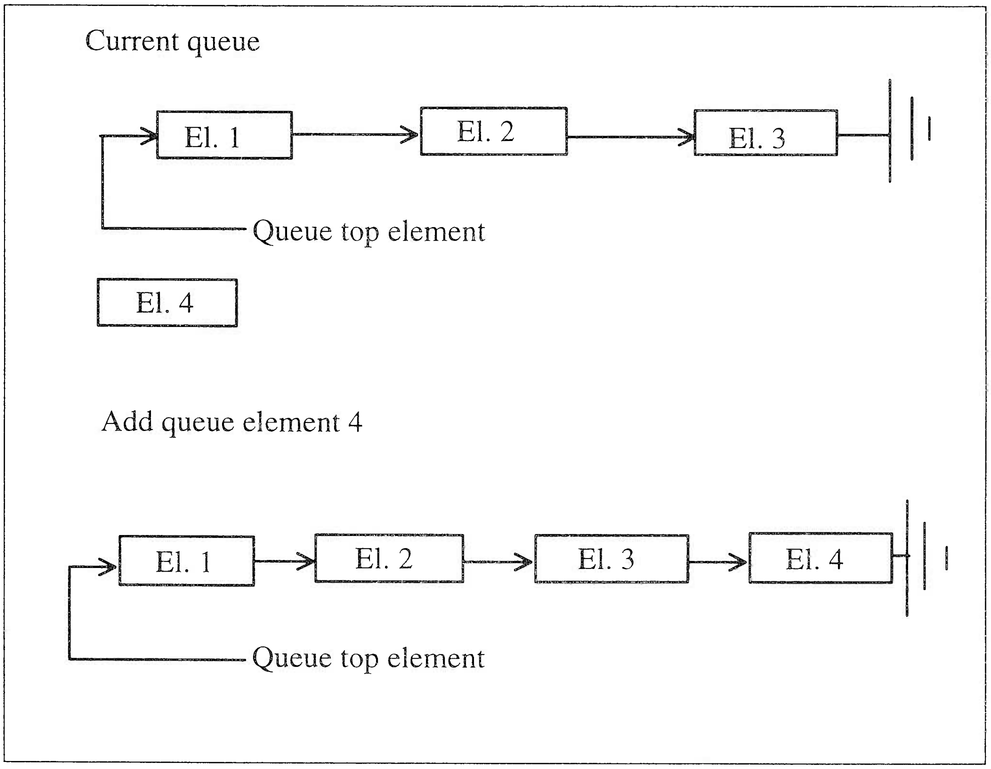


Figure 12-5. *Adding to a queue.*

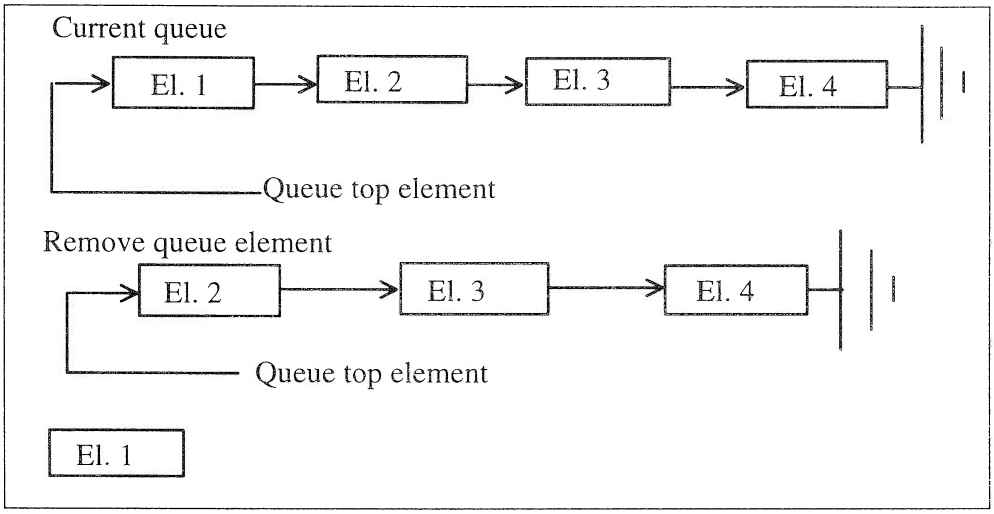


Figure 12-6. *Removing an element from a queue.*

A priority queue data structure is a queue data structure whose elements are ordered by priority. The order is from highest to lowest priority, and the FIFO method is used to order multiple elements with the same priority. Since the items are added to the queue in priority order, removing an element will always remove the top queue element as is depicted in Figure 12-6. Figure 12-7 illustrates adding an item to a priority queue.

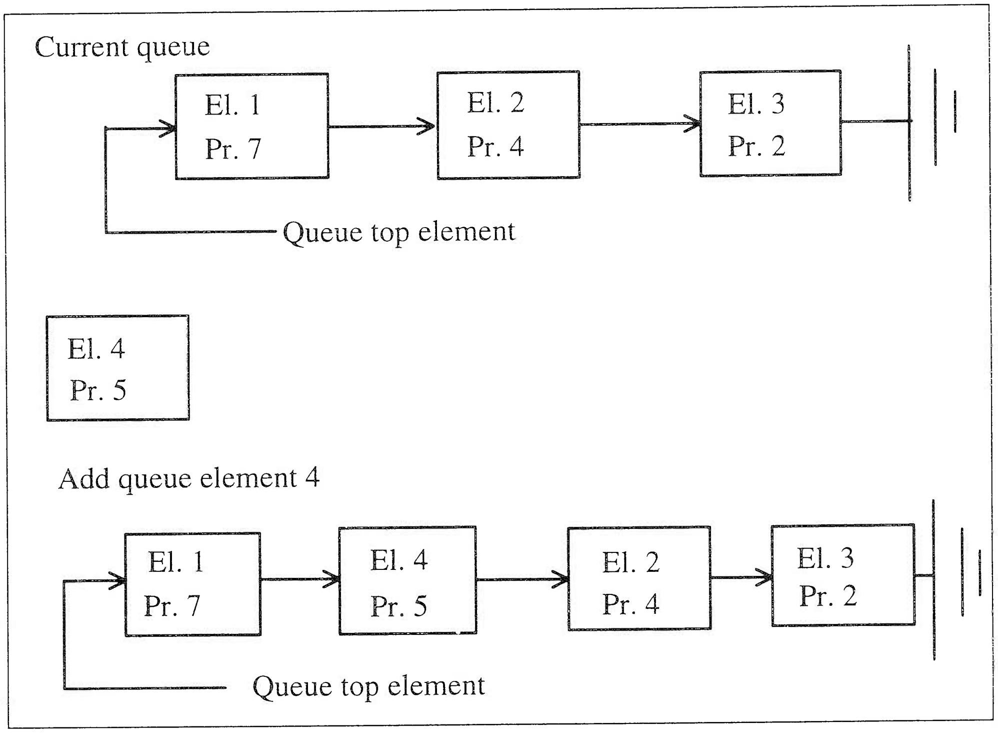


Figure 12-7. *Adding an element to a priority queue.*

FIFO and LIFO ordering is a common need among programs that communicate with queues. With the addition of a priority ordered queue and the ability to override the default ordering used, OS/2 2.1 provides a complete interface to queue element management.

Client and Server Queue Relationship

OS/2 2.1 has a simplified suite of APIs available to manage a queue. These APIs can be used to manage a queue across one or more threads in a single process as well as between multiple processes. All the necessary APIs are available to the server process, which is the process that created the queue. A subset of the APIs can be used from a client process. The client process is any process other than the process that created the queue. Table 12-1 lists those APIs available to the server and the client processes.

SERVER	CLIENT
DosCloseQueue	DosCloseQueue
DosCreateQueue	
	DosOpenQueue
DosPeekQueue	
DosPurgeQueue	
DosQueryQueue	DosQueryQueue
DosReadQueue	
DosWriteQueue	DosWriteQueue

Table 12-1. *Queue APIs available to clients and servers.*

QUEUE APIs

The first step in using a queue is to create one using the DosCreateQueue API. At this point, the queue data structure is indicated along with the queue name. The queue name must be unique within the system. For applications that can have more than one instance of the program running, a common practice is to append the process identifier number to the end of the queue name. Since the process identifier number is assured to be unique within the system, the queue name will also be unique.

DosCreateQueue

```
        DosCreateQueue(
                        PHQUEUE  pphqRWHandle,
                        ULONG    ulQueueFlags,
                        PSZ      pszQueueName);
```

Parameters:

- ☐ pphqRWHandle (PHQUEUE) output—The handle to the queue that is used with the other APIs when referencing this queue.

☐ ulQueueFlags (ULONG) input—Queue attribute flags

QUE_FIFO	(0L)	First In First Out ordered queue.
QUE_LIFO	(1L)	Last In First Out ordered queue.
QUE_PRIORITY	(2L)	Priority ordered queue. Priorities 0 through 15 are supported with 15 being the highest priority.
QUE_NOCONVERT_ADDRESS	(0L)	No conversion of data addresses of elements placed in the queue by 16-bit processes is done.
QUE_CONVERT_ADDRESS	(4L)	Conversion of data addresses from 16-bit addresses to 32-bit addresses is done to elements placed in the queue by 16-bit processes.

☐ pszQueueName (PSZ) input—The name that is associated with the queue. This name must be prefixed with *"\QUEUES\"* and must be unique within the system.

The following example creates a LIFO ordered queue:

```
APIRET rc;
HQUEUE hq;

rc = DosCreateQueue(&hq,                              /* Queue handle */
                    QUE_LIFO,                         /* LIFO ordered */
                    "\\QUEUES\\myLIFOq");             /* queue name   */
```

After the queue has been created, the server process can immediately write elements to the queue. A client process, however, must first open the queue to write elements to it. The DosOpenQueue API must be used to request access to the queue of an existing server process.

DosOpenQueue

```
DosOpenQueue(
             PPID      pppidOwnerPID,
             PHQUEUE   pphqQueueHandle,
             PSZ       pszQueueName);
```

Parameters:

- ☐ pppidOwnerPID (PPID) output—Pointer to the storage location where the process identifier of the server process is returned.
- ☐ pphqQueueHandle (PHQUEUE) output—The handle to the queue that is used with the other APIs when referencing this queue.
- ☐ pszQueueName (PSZ) input—The name that was given to the server process's queue that is to be opened.

Now that the queue has been created or opened, the server or client process has the ability to write an element to the queue. The DosWriteQueue API is used to add an element to a queue.

DosWriteQueue

```
DosWriteQueue(
                HQUEUE  QueueHandle,
                ULONG   ulRequest,
                ULONG   ulDataLength,
                PVOID   pDataBuffer,
                ULONG   ulElemPriority);
```

Parameters:

- ☐ QueueHandle (HQUEUE) input—The handle to the queue that was returned from DosCreateQueue on a server process or from DosOpenQueue on a client process.
- ☐ ulRequest (ULONG) input—A user identifier that can be used as an event indicator by the application.
- ☐ ulDataLength (ULONG) input—The length in bytes of the data buffer whose pointer is passed in parameter pDataBuffer.
- ☐ pDataBuffer (PVOID) input—A pointer to a user-defined buffer of data that is to be associated with the queue element.

- ☐ ulElemPriority (ULONG) input—The priority that is to be given to this queue element. This field has meaning only for queues that were created with the QUE_PRIORITY ulQueueFlags option.

There are three methods for conveying information when an application is writing an element to a queue. The first method is to pass an event indicator in the ulRequest parameter. This parameter is completely defined by the application. Therefore, the application has full control over the meaning of this field when the element is read from the queue. The second method is to specify a priority. Although priorities are restricted to the range 0 through 15, the application has some additional freedom to distinguish both the importance of the queue element and the type of element. The third method is to provide a pointer to a buffer that can be accessed when the element is read from the queue.

This discussion is continued in *Queue Element Data* later in this chapter. Since it is much easier to communicate without passing memory buffers that are dynamically allocated, the ulRequest and ulElemPriority parameters should be utilized first in the design of the queue elements.

After the queue has been created, it can be accessed by any other queue APIs through the queue handle. The server process will typically create the queue on the same thread in which it will reference elements written to the queue. Although not necessary, this provides an isolated part of the program in which this interprocess communication is done. To remove an element from the queue, use the DosReadQueue API.

DosReadQueue

```
DosReadQueue(
             HQUEUE          QueueHandle,
             PREQUESTDATA    pRequest,
             PULONG          pDataLength,
             PPVOID          pDataAddress,
             ULONG           ulElementCode,
             BOOL32          b32NoWait,
             PBYTE           pbElemPriority,
             HEV             SemHandle);
```

Parameters:

- ☐ QueueHandle (HQUEUE) input—The handle to the queue that was created when the application called DosCreateQueue.

☐ pRequest (PREQUESTDATA) output—A pointer to a storage location, allocated by the caller, that contains two fields, one for returning the process identifier of the process that wrote the element to the queue and another for a user-defined field, which can be used as an event code.

☐ pDataLength (PULONG) output—A pointer to a storage location where the length in bytes of the queue element data is returned.

☐ pDataAddress (PPVOID) output—A pointer to a storage location where the pointer to the queue element data is returned.

☐ ulElementCode (ULONG) input—This parameter will be 0 when the application wants to remove the next element from the queue. To read any other element from the queue, this value must be the element code that identifies the queue element that is returned from the DosPeekQueue API.

☐ b32NoWait (BOOL32) input—This parameter specifies the action that is to be performed if no element is present on the queue.

```
DCWW_WAIT    (0)    The calling thread waits until something
                    is found on the queue.

DCWW_NOWAIT (1)     The API returns immediately, if no
                    element is present, with the
                    ERROR_QUE_EMPTY return code.
```

☐ pbElemPriority (PBYTE) output—The pointer to the storage location where the queue elements priority value can be returned. A value between 0 and 15 can be passed on the DosWriteQueue API for queues that were created with the QUE_PRIORITY option.

☐ SemHandle (HEV) input—An optional event semaphore handle that will be posted when an element is written to the queue. This option can be used only with the DCWW_NOWAIT option.

The b32NoWait parameter offers an option to wait for an element to be written to the queue, when using the DCWW_WAIT flag. This option is appropriate if the calling thread has no other work that it needs to do. The program in Figure 12-8 illustrates this method.

```
/*  prog12f8.c                                                          */
/*  Waiting on a write to a queue.                                      */

#define INCL_DOSPROCESS
#define INCL_DOSQUEUES
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>

#define STACKSIZE 4096
#define KILLQUEUE 1000

VOID WriteQueue(PVOID arglist);
HQUEUE hqQueue;

INT main(VOID)
{
   REQUESTDATA reqData;
   PVOID pMsg;
   ULONG cbLen, ulItem = 1;
   BYTE Prio;

   DosCreateQueue(&hqQueue, QUE_FIFO, "\\QUEUES\\myqueue");

   _beginthread(WriteQueue, NULL, STACKSIZE, (PVOID)NULL);

   /* Wait until an element is found on the queue and                  */
   /* then read it from the queue.                                      */

   while (!DosReadQueue(hqQueue, &reqData, &cbLen,
                        &pMsg, 0, DCWW_WAIT, &Prio,
                        (HEV)NULL)) {
      printf("Reading element #%d, ulUser %d\n", ulItem++,
             reqData.ulData);
      if (reqData.ulData == KILLQUEUE) {
         break;
      }
   }

   DosCloseQueue(hqQueue);
   return(0L);
}

VOID WriteQueue(PVOID arglist)
{
   USHORT i;
   ULONG item;

   for (i = 0; i < 10; ++i) {
      if (i != 9) {
         item = i;
      } else {
         item = KILLQUEUE;
      }
```

```
         printf("Writing element %d\n", item);
         DosWriteQueue(hqQueue, item, 0, (PSZ)NULL, 0);
         DosSleep(1);
      }
}
```

Figure 12-8. *Program illustrating waiting on a queue for something to be written to it.*

The b32NoWait parameter also offers an option not to wait if there is no element on the queue, when using the DCWW_NOWAIT flag. In this case, the API returns ERROR_QUE_EMPTY. However, an event semaphore can be associated with the queue by passing its handle in the SemHandle parameter. The semaphore will be posted when an element is added to the queue.

After a semaphore is associated with the queue, the system remembers the semaphore handle. Only that semaphore handle can be associated with the queue. If any other semaphore is passed, ERROR_INVALID_PARAMETER is returned. With an event semaphore associated with a queue, an application can wait for just that semaphore or for multiple event semaphores to be posted. The program in Figure 12-9 illustrates this method.

```
/*  prog12f9.c                                                    */
/*  Waiting on an event semaphore.                                */

#define INCL_ERRORS
#define INCL_DOSSEMAPHORES
#define INCL_DOSPROCESS
#define INCL_DOSQUEUES
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>

#define STACKSIZE 4096
#define KILLQUEUE 1000

VOID WriteQueue(PVOID arglist);
HQUEUE hqQueue;

INT main(VOID)
{
   REQUESTDATA reqData;
   PVOID pMsg;
   ULONG cbLen, ulItem = 1;
   BYTE Prio;
   HEV hevMySem;
   DosCreateQueue(&hqQueue, QUE_FIFO, "\\QUEUES\\myqueue");

   DosCreateEventSem((PSZ)NULL, (PHEV)&hevMySem,
                     0, FALSE);
   DosReadQueue(hqQueue, &reqData, &cbLen, &pMsg, 0,
                DCWW_NOWAIT, &Prio, (HEV)hevMySem);

   _beginthread(WriteQueue, NULL, STACKSIZE, (PVOID)NULL);

   while (!DosWaitEventSem(hevMySem, SEM_INDEFINITE_WAIT)) {
      while (DosReadQueue(hqQueue, &reqData, &cbLen,
                          &pMsg, 0, DCWW_NOWAIT, &Prio,
                          (HEV)hevMySem)
             != ERROR_QUE_EMPTY) {
         printf("Reading element #%d, ulUser %d\n", ulItem++,
                reqData.ulData);
         if (reqData.ulData == KILLQUEUE) {
            break;
         }
      }
      if (reqData.ulData == KILLQUEUE) {
         break;
      }
   }

   DosCloseEventSem(hevMySem);
   DosCloseQueue(hqQueue);
   return(0L);
}

VOID WriteQueue(PVOID arglist)
{
   USHORT i;
   ULONG item;

   for (i = 0; i < 10; ++i) {
      if (i != 9) {
         item = i;
      } else {
         item = KILLQUEUE;
      }
      printf("Writing element %d\n", item);
      DosWriteQueue(hqQueue, item, 0, (PSZ)NULL, 0);
      DosSleep(1);
   }
}
```

Figure 12-9. *Program illustrating waiting on an event semaphore that gets posted when an element is written to a queue.*

Similar to the DosReadQueue API, the DosPeekQueue API allows you to reference information associated with queue elements. However, it does not remove the element from the queue. The DosPeekQueue can either examine elements as they are ordered on the queue or examine an element that is found after an element on the queue. The DosPeekQueue API is typically used only on the thread that issues the DosReadQueue in the server process.

Otherwise, some form of synchronization will be required to prevent the thread that issues the DosReadQueue from reading the element from the queue. If the read was to occur before the processing of the element information was completed by the thread that issued the DosPeekQueue, the program might process the single element twice or access element data that no longer exists.

DosPeekQueue

```
DosPeekQueue(
            HQUEUE          QueueHandle,
            PREQUESTDATA    pRequest,
            PULONG          pDataLength,
            PPVOID          pDataAddress,
            PULONG          pElementCode,
            BOOL32          b32NoWait,
            PBYTE           pbElemPriority,
            HEV             SemHandle);
```

Parameters:

- ☐ QueueHandle (HQUEUE) input—The handle to the queue that was created when the application called DosCreateQueue.

- ☐ pRequest (PREQUESTDATA) output—A pointer to a storage location, allocated by the caller, that contains two fields, one for returning the process identifier of the process that wrote the element to the queue and another for a user-defined field which can be used as an event code.

- ☐ pDataLength (PULONG) output—A pointer to a storage location where the length in bytes of the queue element data is returned.

- ☐ pDataAddress (PPVOID) output—A pointer to a storage location where the pointer to the queue element data is returned.

- ☐ pElementCode (PULONG) input and output—A pointer to a storage location the queue element code is returned to. If this storage location is initialized to 0, the top element in the queue is peeked. If the location is an element code from a previous call to DosPeekQueue, the queue element that is peeked is the one following that element.

- ☐ b32NoWait (BOOL32) input—This parameter specifies the action that is to be performed if no element is present on the queue.

```
DCWW_WAIT   (0)  The calling thread waits until something
                 is found on the queue.

DCWW_NOWAIT (1)  The API returns immediately, if no
                 element is present, with the
                 ERROR_QUE_EMPTY return code.
```

- ☐ pbElemPriority (PBYTE) output—The pointer to the storage location where the queue element's priority value can be returned. A value between 0 and 15 can be passed on the DosWriteQueue API for queues that were created with the QUE_PRIORITY option.

- ☐ SemHandle (HEV) input—An optional event semaphore handle that will be posted when an element is written to the queue. This option can be used only with the DCWW_NOWAIT option.

Except for the fact that the DosPeekQueue API doesn't remove the element from the queue, the only other difference between it and the DosReadQueue API is the chance to use the pElementCode parameter.

This parameter is used to help traverse the elements on the queue. This allows the program the option of searching for a particular queue element. Once found, the program can then issue the DosReadQueue API to read that particular element instead of the next ordered element.

Both the server and the client processes should close the queue when it is no longer required. The DosCloseQueue API should be issued once for the server process and once for every DosOpenQueue called on any client process. After the queue has been closed on a server process, the client process can no longer use the queue.

Any attempt by a client process to use the queue handle that is returned from the DosOpenQueue API, after the server process has closed that queue, will result in the ERROR_QUE_INVALID_HANDLE return code.

A client process's use of a queue will terminate once an equal number of calls to DosOpenQueue and DosCloseQueue are issued.

DosCloseQueue

```
DosCloseQueue(
                HQUEUE QueueHandle);
```

Parameter:

- ☐ QueueHandle (HQUEUE) input—The handle to the queue that was created on a call to DosCreateQueue or opened on a call to DosOpenQueue.

The DosPurgeQueue API can be used by a server process to remove all elements from a queue.

DosPurgeQueue

```
DosPurgeQueue(
                HQUEUE  QueueHandle);
```

Parameter:

- ☐ QueueHandle (HQUEUE) input—The handle to the queue that was created when the application called DosCreateQueue.

The DosQueryQueue API is available to both server and client processes to query the number of elements that are on the queue.

DosQueryQueue

```
DosQueryQueue(
                HQUEUE  QueueHandle,
                PULONG  pNumberElements);
```

Parameters:

- ☐ QueueHandle (HQUEUE) input—The handle to the queue that was created on a call to DosCreateQueue or opened on a call to DosOpenQueue.

- ☐ pNumberElements (PULONG) output—A pointer to a location where the number of queue elements can be returned.

QUEUE ELEMENT DATA

As was mentioned during the discussion on the DosWriteQueue API, there are three methods of conveying information with a queue element. Two methods are through using the ulRequest and ulElemPriority parameters on the DosWriteQueue API. The third is to pass a pointer in the pDataAddress parameter. This pointer can point to any size of buffer, and the buffer size can be passed in the ulDataLength parameter.

Because writing and reading a queue are not done synchronously, the pointer that is passed must be one that references a buffer that will still exist after the element is read from the queue. Because of this requirement, the buffer is typically dynamically allocated. The program in Figure 12-10 illustrates this approach.

```
/   prog12f10.c                                                      */
/*  Passing a data pointer.                                          */

#define INCL_DOSPROCESS
#define INCL_DOSQUEUES
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>
#define STACKSIZE 4096
#define KILLQUEUE 1000

VOID WriteQueue(PVOID arglist);
HQUEUE hqQueue;

INT main(VOID)
{
   REQUESTDATA reqData;
   PVOID pMsg;
   PSZ string;
   ULONG cbLen, ulItem = 1;
   BYTE Prio;

   DosCreateQueue(&hqQueue, QUE_FIFO, "\\QUEUES\\myqueue");

   _beginthread(WriteQueue, NULL, STACKSIZE, (PVOID)NULL);

   /* Wait until an element is found on the queue and                */
   /* then read it from the queue.                                   */
   while (!DosReadQueue(hqQueue, &reqData, &cbLen,
                        &pMsg, 0, DCWW_WAIT, &Prio,
                          (HEV)NULL)) {
      string = (PSZ)pMsg;
      printf("Reading element #%d string \"%s\"\n", ulItem++,
             string);
      free(string);
      if (reqData.ulData == KILLQUEUE) {
         break;
      }
   }

   DosCloseQueue(hqQueue);
   return(0L);
}


VOID WriteQueue(PVOID arglist)
{
   USHORT i;
   ULONG item;
   PSZ str;

   for (i = 0; i < 10; ++i) {
      if (i != 9) {
         item = i;
      } else {
         item = KILLQUEUE;
      }
      str = (PSZ)malloc(15);
      strcpy(str, "dynamic buffer");
      printf("Writing element %d\n", item);
      DosWriteQueue(hqQueue, item, strlen(str), (PVOID)str,
                    0);
      DosSleep(1);
   }
}
```

Figure 12-10. *Program illustrating passing a data pointer with the data written with a queue element.*

This program usesthe malloc and free dynamic memory allocation routines to get the buffer that was used to pass a string along with the queue element. This method will work for passing buffer pointers within the same process. However, a shared memory approach will be necessary when multiple processes are involved.

Notice also that the thread performing the reading from the queue is responsible for freeing the buffer. This is a typical method used for managing buffers whose pointers to them are passed asynchronously.

SINGLE THREAD MULTIPLE QUEUE MANAGEMENT

For various reasons, such as supporting internally and externally known queues, an application design might need to wait on more than one queue on a single thread. As you have seen earlier in this chapter, the DosReadQueue supports waiting on only one queue. Therefore, at a minimum, each queue needs an event semaphore associated with it so a program can wait on one queue for a while and then switch to another queue after a particular interval of time has elapsed.

The problem with this design is that your thread is busy jumping from one queue to the next even though there may not be anything to be found on the queue during certain periods. Therefore, this amounts to a polling activity which wastes system resources. The solution is to use a muxwait semaphore that can be waited on until any one of the event semaphores is posted. The posting of a semaphore occurs when a DosWriteQueue API is called. The program in Figure 12-11 illustrates this solution.

```
/*  prog12f11.c                                                   */
/*  Waiting on multiple queues.                                   */

#define INCL_ERRORS
#define INCL_DOSPROCESS
#define INCL_DOSSEMAPHORES
#define INCL_DOSQUEUES
#include <os2.h>
#include <stdlib.h>
#include <stdio.h>

#define QCOUNT 4
#define STACKSIZE 4096

VOID WriteQueue(PVOID arglist);

INT main(VOID)
{
   SEMRECORD hevMyEvents[QCOUNT];
   REQUESTDATA reqData;
   CHAR qname[20], qnamesuf[5];
   HQUEUE qa[QCOUNT];
   USHORT cWrites[QCOUNT];
   ULONG i, ulUser, cbLen, cPosts;
   PVOID pMsg;
   HMUX hmuxMyMuxwait;
   BOOL bRaceOver;
   BYTE Prio;
   /* Create QCOUNT number of queues and event semaphores         */
   /* and associate the event semaphore with the queue            */
   for (i = 0; i < QCOUNT; ++i) {

      /* Create a unique queue name                               */
      memset(&qname, '\0', sizeof(qname));
      strcpy(qname, "\\queues\\multq");
      memset(qnamesuf, '\0', sizeof(qnamesuf));
      _itoa(i, qnamesuf, 10);
      strcat(qname, qnamesuf);

      DosCreateQueue(&qa[i], QUE_FIFO, qname);
      DosCreateEventSem((PSZ)NULL,
                        (PHEV)&hevMyEvents[i].hsemCur,
                        0, FALSE);
      hevMyEvents[i].ulUser = i;
      DosReadQueue(qa[i], &reqData, &cbLen, &pMsg, 0,
                   DCWW_NOWAIT, &Prio,
                   (HEV)hevMyEvents[i].hsemCur);
   }

   /* Create the muxwait semaphore so you can wait on             */
   /* any of the event semaphore getting posted.                  */
   DosCreateMuxWaitSem((PSZ)NULL, &hmuxMyMuxwait, QCOUNT,
                       hevMyEvents, DCMW_WAIT_ANY);

   /* Create the QCOUNT number of threads that will at            */
   /* random times write an element to the queue.                 */
   for (i = 0; i < QCOUNT; ++i) {
      _beginthread(WriteQueue, NULL, STACKSIZE,
                   (PVOID)qa[i]);
   }

   /* Loop until one of the queues has been written to            */
   /* QCOUNT times.                                               */
   bRaceOver = FALSE;
   memset(cWrites, '\0', sizeof(cWrites));
   while (!bRaceOver) {

      /* Wait until one of the queues is written to               */
      DosWaitMuxWaitSem(hmuxMyMuxwait, SEM_INDEFINITE_WAIT,
                        &ulUser);

      /* Read everything off the queue, increment the             */
      /* count for this queue and reset the event sem.            */
      while (DosReadQueue(qa[ulUser], &reqData, &cbLen,
                          &pMsg, 0, DCWW_NOWAIT, &Prio,
                          (HEV)hevMyEvents[ulUser].hsemCur)
             != ERROR_QUE_EMPTY) {
         cWrites[ulUser]++;
         printf("Reading from Queue #%d, write #%d\n", ulUser,
                cWrites[ulUser]);
      }
```

```
         DosResetEventSem((HEV)hevMyEvents[ulUser].hsemCur,
           &cPosts);

         if (cWrites[ulUser] == QCOUNT) {
            bRaceOver = TRUE;
         }
      }

      /* Destroy the queue and semaphore resources this              */
      /* program used.                                               */
      for (i = 0; i < QCOUNT; ++i) {
         DosCloseQueue(qa[i]);
         DosCloseEventSem((HEV)hevMyEvents[i].hsemCur);
      }
      DosCloseMuxWaitSem(hmuxMyMuxwait);
      return(0L);
   }

   VOID WriteQueue(PVOID arglist)
   {
      USHORT i;
      APIRET rc;
      ULONG x;

      /* Issue QCOUNT number of writes to the queue whose            */
      /* handle was passed when creating the thread at               */
      /* random intervals of time less than a second.                */
      for (i = 0; i < QCOUNT; ++i) {
         x = rand();
         DosSleep(x % 1000);
         rc = DosWriteQueue((HQUEUE)arglist, 0, 0, (PSZ)NULL,
                            0);

         /* If the handle is invalid the program must have           */
         /* closed the queue therefore fall out of the               */
         /* thread.                                                  */
         if (rc == ERROR_QUE_INVALID_HANDLE) {
            break;
         }
      }
   }
```

Figure 12-11. *Program illustrating waiting on multiple queues.*

The program in Figure 12-11 generates the following output:

```
Reading from Queue #3, write #1
Reading from Queue #0, write #1
Reading from Queue #1, write #1
Reading from Queue #2, write #1
Reading from Queue #0, write #2
Reading from Queue #1, write #2
Reading from Queue #2, write #2
Reading from Queue #3, write #2
Reading from Queue #3, write #3
Reading from Queue #0, write #3
Reading from Queue #1, write #3
Reading from Queue #2, write #3
Reading from Queue #0, write #4
```

Figure 12-12. *Output from the program in Figure 12-11.*

The program in Figure 12-11 simply creates QCOUNT number of queues and associates an event semaphore with each of them. Once created, a muxwait semaphore waits on one of them to get written to. When a queue gets written to, all the items are read from the queue and the queue's event semaphore is reset. A thread is created per queue and at random time intervals writes to the queue. The program ends when one queue has QCOUNT number of elements written to it.

SUMMARY

Queues are a key tool that is available for interprocess communication. Queues are often used as the primary form for a program's event management and interprocess communication. This chapter focuses on all aspects available with OS/2 2.1 queues. Aspects including element order, element data, and element reading and writing are fully discussed. Also the following OS/2 APIs are prominently discussed:

- ☐ DosCloseQueue—This API is used to close access to a queue by either a client or a server process.
- ☐ DosCreateQueue—This API is used to create a queue. Once a queue has been created, elements can be written to the queue by the server process.
- ☐ DosOpenQueue—This API is used by a client process to gain authority to write elements to a server process's queue.
- ☐ DosPeekQueue—This API is used to look at an element on the queue and is available only to the server process.
- ☐ DosPurgeQueue—This API allows the server process to remove all elements from its queue.

☐ DosQueryQueue—This API returns the number of elements that currently exist on the queue and is available to both the server and the client processes.

☐ DosReadQueue—This API is used to read off an element from the queue by the server process.

☐ DosWriteQueue—This API is used to write an element to a queue and is available to both the server and the client processes.

CHAPTER 13

Timers

"But at my back I always hear
Time's wingèd chariot hurrying near."
—Andrew Marvell

INTRODUCTION

People use measurements of time in many different ways. You use a single, absolute value such as Monday or 9:00 AM to indicate what time a meeting is scheduled or when a project is due. You use a relative time to indicate how long a task will take, for example, a program will take six hours to write. You use a different relative time to indicate when a task will be finished, for example, that same program may be available only after 12 hours, because other tasks must also be done during that time.

These three types of measurements are sometimes necessary in a computer program. This chapter will explore how these concepts are used in a computer system and show how to use the APIs associated with measuring time in OS/2 2.1.

THE EYE OF THE BEHOLDER—RELATIVE TIMES

How long does it take for a program to run anyway? That depends. OS/2 2.1 is a multitasking, preemptive operating system. This means that several applications can run simultaneously: One runs for a while, then it's suspended while another runs for a while, then the first may run again, and so on. People run into this in real life all the time. For example, you're cooking dinner and one of your children starts crying, or you're watching television and the phone rings.

Suppose you are an end user waiting for your job to finish. If you start the job at 9:30 and it finishes at 9:45, the job took fifteen minutes to run. But if there were other jobs running simultaneously, and your job was suspended for ten of those fifteen minutes, then your job really only took five minutes to run. If you have a customer on the phone, the fifteen minutes counts; if you are paying for computer time, only the five minutes should count.

In computer terminology, the fifteen-minute interval is called *real time*, and the five-minute interval is called *CPU time*. You may be familiar with the term *connect time*, which is analogous to real time.

THE EYE OF OS/2 2.1—ABSOLUTE TIMES AND CLOCKS

So, what about OS/2 2.1? OS/2 2.1 uses the concept of time for itself, to determine when to swap out one process for another. It also uses the concept of time to allow you to tune your applications based on what the human user will see.

There are three clocks you should know about: The *machine cycle clock*, the *millisecond counter/clock*, and the system's *date/time clock*. The machine cycle clock is not really a clock at all. Inside the computer, an electronic pulse is generated at regular intervals, with a machine cycle being the length of real time between pulses. When you see the speed of a chip stated as 33MHz, for example, it is referring to the number of pulses per second. The faster the pulses are generated, the faster the computer is.

The second clock is the millisecond counter. This is not really a clock either. As you probably guessed from its name, it is a counter. OS/2 2.1 sets the counter to 0 when it is booted and adjusts the counter every few milliseconds.

This clock is the one you should focus on when you are considering the accuracy of all of the time intervals talked about later in this chapter. You can read the value of the millisecond counter by using the DosQuerySysInfo API. You can determine how often the counter is updated by using other information from the DosQuerySysInfo API. For example, on a 20MHz PS/2 Model 80, the clock ticks every 310 tenths of a millisecond, which is approximately 32 times a second.

The third clock is really a clock. Well, sort of. As far as we're concerned, it's a digital clock. You can use it to determine the current day, date, and time, as well as some other information related to date and time.

In general you will be able to measure only real time; the only way to determine CPU time using these clocks is by ensuring that your application is not interrupted or preempted in between the timings.

ACCURACY

To understand how accurate time is within a computer, you must understand about clock ticks. A *clock tick* is a fixed interval of time, at the end of which an interrupt is generated and the computer increments its counters and system clock to note that a fixed amount of time has gone by. This tick will correspond to a fixed number of milliseconds (or nanoseconds), so OS/2 2.1 can use it to calculate real time.

The granularity of time measurements is determined by the length of the clock tick. The faster the clock, the finer the granularity can be. Since the millisecond clock is updated only after every clock tick, the length of the clock tick is the smallest unit of time measurable by the operating system.

You already know how to determine what that unit of time is. Since this unit determines the granularity of the time measurements, it will play a major role in determining how accurate your times are.

When you request a time interval, two factors will affect how close the actual interval comes to the requested interval. One factor is how close the interval is to a multiple of the granularity of the internal clock. The other is multitasking.

Consider the granularity issue first. Since the operating system can accurately measure only the times that are multiples of the clock tick, OS/2 2.1 will round the time of your request up, if necessary.

As an example, if the clock ticks once every half-second, then OS/2 2.1 cannot distinguish between a quarter-second and a half-second. If you request a time interval of a quarter-second, your request will be rounded up to a half-second.

Now consider the second factor, multitasking. Since OS/2 2.1 is a preemptive multitasking system, it may be executing a different application when your time interval ends. (You are making a 3-minute soft-boiled egg and the phone rings. While you go to answer it, the 3 minutes end. By the time you get back to the egg, 3 minutes 20 seconds may have gone by. You didn't remove the egg after exactly three minutes because you were busy with the phone.) In this case, your application will be scheduled to run again but will not actually begin running at the exact time that the requested time interval ends.

You can illustrate this quite dramatically by writing a program and starting it in two different windows simultaneously. If you start them and then click the mouse in a third window, the two windows with your program running will probably finish at the same time. However, since OS/2 2.1 gives priority to the window with the focus, starting the programs and leaving the focus in one of them (to do this, don't move the mouse after starting the second window) will cause that window to finish much sooner than the other one.

If your application requires stricter timing, you will need to use more than just the timer. You will need to use priorities and interrupts, which are covered in Chapters 1 and 9 in this book. By setting a high priority on your application's thread, you can inhibit being preempted.

DATES AND TIMES

OS/2 2.1 maintains information regarding the current date and time. This value is initialized when the operating system is brought up and can be changed at any time by either the user or an application.

To set or change the current date or time, the user can type in a value, or an application can set an arbitrary value. If the machine is attached to a network, an application can also find the date and time from another machine. OS/2 2.1 can then easily track passing time using its internal clock (and the clock ticks discussed earlier). The date and time APIs use a DATETIME structure, which has the following fields:

- ❑ Hours (UCHAR)—Current hour, ranging from 0 to 23.
- ❑ Minutes (UCHAR)—Current minutes past the hour, ranging from 0 to 59.

- ❑ Seconds (UCHAR)—Current seconds past the minute, ranging from 0 to 59.
- ❑ Hundredths (UCHAR)—Current hundredths of a second past the second, ranging from 0 to 99.
- ❑ Day (UCHAR)—Current day of the month, ranging from 1 to 31. The number of days in a month is based on the Julian calendar.
- ❑ Month (UCHAR)—Current month of the year, ranging from 1 to 12. The months are based on the Julian calendar.
- ❑ Year (USHORT)—Current year. There are always 12 months in a year, and either 365 or 366 days.
- ❑ TimeZone (SHORT)—The difference in hours between the current time zone and Greenwich Mean Time. For example, the value for Central Standard Time is 6. This field is useful for distributed applications.
- ❑ DayofWeek (UCHAR)—Current day of the week, ranging from 0 to 6, with Sunday being 0.

All of the UCHAR fields in the DATETIME structure are actually 1 byte of numeric data, not character data. That is, if it is 8 PM (20:00), the hexadecimal value of the Hours field would be 0x14. If it is a Thursday (day 4), the hexadecimal value of the Day field would be 0x04, not 0x34 (0x34 is the ASCII representation of the character 4). The main reason for this is to allow calculations with the fields. A secondary reason is that an integer is never dependent on code pages. No matter what country code is in use, the structure always has the same meaning. This is particularly important in double byte code pages. The double byte character set (DBCS) is used for Japanese, Korean, and Chinese translations. The fields in the structure are also defined as UCHAR to save space, since an INT field is at least 2 bytes.

The two APIs for dates and times are DosSetDateTime and DosGetDateTime.

DosSetDateTime

```
DosSetDateTime(
                 PDATETIME  pDateTime);
```

Parameter:

- ❑ pDateTime (PDATETIME) input—A pointer to the structure which contains the date and time information as discussed previously.

This function sets the current date and time known to OS/2 2.1 to be that contained in the structure. The current date and time are determined by how many clock ticks have passed since the last time the DosSetDateTime function was called. Note that when you use the command line to set a new date or time, this function is being called.

DosGetDateTime

```
DosGetDateTime(
                    PDATETIME  pDateTime);
```

Parameter:

- ❑ pDateTime (PDATETIME) input and output—A pointer to a structure in which to put the date and time information.

This function places the current date and time into the structure. The caller must allocate space for the structure.

Figure 13-1 is a simple program that sets the system's date and time to 1 PM Central Standard Time on April 1, 1993.

```
/*                                                           */
/*  prog13f1.  Setting the system clock.                     */
/*                                                           */

#define INCL_DOS
#include <os2.h>

int main (int argc, void *argv[])
{

   DATETIME DateTime;                       /* DATETIME structure */
   /* First, set up the structure                            */
   /* 1 PM is 13 hours, 0 minutes                            */
   DateTime.Hours =  (UCHAR) 13;
   DateTime.Minutes =       (UCHAR) 0;
   DateTime.Seconds =       (UCHAR) 0;
   DateTime.Hundredths = (UCHAR) 0;
   /* April 1, 1993 is day 1, month 4, year 1993                        */
   DateTime.Day =    (UCHAR) 1;
   DateTime.Month =  (UCHAR) 4;
   DateTime.Year =  (USHORT) 1993;

   /* CST is 6 hours off GMT                                            */
   DateTime.TimeZone =     (SHORT) 6;

   /* April 1, 1993 is a Thursday                                       */
   DateTime.DayofWeek =  (UCHAR) 4;

   /* Now we can use the API to set the current date                    */
   rc = DosSetDateTime(&DateTime);

   return 0;
}
```

Figure 13-1. *Setting the system clock.*

Figure 13-2 is a program that compares a month and day to the current month and day and prints a message if they are equal.

```
/*                                                                      */
/*  prog13f2.  Using the system clock.                                  */
/*                                                                      */

#define INCL_DOS
#include <os2.h>

int main (int argc, void *argv[])
{
   /* This program has two parameters, the first is a month            */
   /* and the second is a day                                           */

   DATETIME DateTime;                      /* DATETIME structure        */
   int     BirthMonth;
   int     BirthDay;

   BirthMonth = atoi((PSZ) argv[1]);
   BirthDay = atoi((PSZ) argv[2]);

   DosGetDateTime(&DateTime);              /* Get the current date */
   /* Compare the current date to the input day and month               */
   if ((DateTime.month == (UCHAR) BirthMonth) &&
       (DateTime.day   == (UCHAR) BirthDay)){
   }    printf("Happy Birthday!\n");
   else if (DateTime.month == BirthMonth){
       if (DateTime.day > BirthDay)
        printf("Happy Belated Birthday!\n");
      else
        printf("Your birthday is this month!\n");
   }  else if (DateTime.month < BirthMonth)    {
      printf("Your birthday is in another %d months.\n",
            BirthMonth - DateTime.month);
   } else  {
      printf("Your birthday is in another %d months.\n",
            BirthMonth + 12 - DateTime.month);
   }
   return 0;
}
```

Figure 13-2. *Using the system clock.*

TIMED INTERVALS

Two types of APIs handle timed intervals. One uses CPU-time; the other, real time.

An API that Uses CPU-Time

Suppose you are writing a multi-user application in which users may occasionally attempt to access a resource, such as as file, simultaneously. When that happens, you want each user to wait for several seconds before trying again. For example, the Ethernet communications protocol allows only one message to be on the Ethernet at a time. The protocol uses this method, along with a random number generator, to handle collisions among applications trying to send messages.

There are several ways to handle collisions. The simplest one, shown in Figure 13-3, uses the DosSleep API. Unlike other APIs, the time specified in this API is CPU-time. Recall that in OS/2 2.1, each application is scheduled to use the CPU for some amount of time before it may have to give up the CPU to another application. In the DosSleep API, the time given is the cumulative *scheduled* time that must pass before your application is allowed to run again. This method adds an extra degree of randomness to the wait. Figure 13-3 introduces the DosSleep API.

DosSleep

```
        DosSleep(
                ULONG lengthoftime);
```

Parameter:

- ❑ lengthoftime (ULONG) input—How long the program should be suspended, in milliseconds. In this case it is a measure of CPU-time, not real time.

```
/*                                                            */
/*  prog13f3.  DosSleep                                              */
/*                                                                   */

#define INCL_DOS
#include <os2.h>

int main (int argc, void *argv[])
{
   /* This program sleeps for 2 seconds of scheduled time            */

   ULONG sleepTime;

   sleepTime = 2000;                /* 2 seconds, in milliseconds */
   DosSleep(sleepTime);             /* Sleep for 2 seconds          */
   printf("Yaaawwwn.  That was a nice nap!\n");

   return 0;
}
```

Figure 13-3. *The DosSleep API.*

APIs that Use Real Time

Suppose you have a computer that runs the appliances in your home, and you want to be able to set an alarm, say, for turning off the oven. You could use the DosSleep API, as in Figure 13-3, but that would not give very good results if there were many applications running simultaneously (since it uses CPU-time, not real time). You would end up overcooking the meal, and while you might go out to eat, you'd still have to do the dishes when you got back.

The best way to set this kind of alarm is to use a one-shot timer. The DosAsyncTimer API makes use of a shared event semaphore (semaphores are fully described in Chapter 10). When the requested time has elapsed, the semaphore will be posted. With this API, you have two choices. By waiting on the semaphore, you can suspend all processing until the time has elapsed. Or, you can dedicate a thread to wait on the semaphore and continue processing other tasks.

Figure 13-4 shows a sample program using DosAsyncTimer. For simplicity, we do not spawn a separate thread to wait on the timer. You should always check the return code from this call, since an error will cause it to return before the specified time has elapsed.

Figure 13-4 introduces the DosAsyncTimer API.

DosAsyncTimer

```
          DosAsyncTimer(
                        ULONG    ulLengthOfTime,
                        HSEM     hsemEventSem,
                        PHTIMER  Timer);
```

Parameters:

- ❑ lengthoftime (ULONG) input—How much real time should elapse, in milliseconds, before posting the semaphore.
- ❑ hsemEventSem (HSEM) input—Handle of the event semaphore that should be posted when the time has passed; should be reset before starting the timer.
- ❑ Timer (PHTIMER) output—Pointer to the handle for the timer. You can use this handle in another thread to stop the timer before the time has elapsed, using the DosStopTimer API.

```
/*                                                          */
/*  prog13f4.  DosStopTimer.                                */
/*                                                          */

#define INCL_DOS
#include <os2.h>

int main (int argc, void *argv[])
{
   /* This program sounds an alarm after the number         */
   /* of seconds input to the program.                      */

   HEV    ESem;
   HTIMER Timer;
   ULONG  AlarmTime;
   int    rc;

   /* Convert input to milliseconds                         */
   AlarmTime  = (ULONG) (1000 * atoi((PSZ) argv[1]));

   /* Create a shared, unnamed event semaphore              */
   DosCreateEventSem (NULL,                /*  unnamed       */
                      &ESem,               /*  ptr to handle */
                      DC_SEM_SHARED,       /* shared         */
                      0);                  /* initially 'set' */
   rc = DosAsyncTimer(AlarmTime, (HSEM) ESem, &Timer);
   if (rc) {
      printf("DosAsyncTimer failed.  RC = %ld\n", rc);
   }
   DosWaitEventSem(ESem, SEM_INDEFINITE_WAIT);
   DosBeep(1300, 0500);
   return 0;
}
```

Figure 13-4. *The DosAsyncTimer API.*

Suppose you want to sound an alarm every 10 minutes. You can't use DosSleep for the same reason you couldn't use it in Figure 13-4; DosSleep does not deal with real time. You could use DosAsyncTimer, but you would have to create a new timer every 10 minutes. The best method for setting a repeatable alarm (say, for a snooze function on an alarm clock) is to use the DosStartTimer API.

The DosStartTimer API is very similar to the DosAsyncTimer API. The difference is that the associated event semaphore will be posted every time the specified time has passed. If you were unable to process anything until the time interval had gone by twice, the semaphore would have been posted twice, so you would know about it.

Figure 13-5 shows how to sound an alarm every 5 seconds for 30 seconds. Two APIs are introduced:

DosStartTimer

```
DosStartTimer(
                   ULONG ulLengthOfTime,
                   HSEM hsemEventSem,
                   PHTIMER Timer);
```

Parameters:

- ❑ lengthoftime (ULONG) input—How much real time should elapse, in milliseconds, before posting the semaphore.
- ❑ hsemEventSem (HSEM) input—Handle of the event semaphore that should be posted when the time has passed; should be reset before starting the timer.
- ❑ Timer (PHTIMER) output—Pointer to the handle for the timer. You can use this handle in another thread to stop the timer before the time has elapsed, using the DosStopTimer API.

DosStopTimer

```
          DosStopTimer(
                       HTIMER Timer);
```

Parameters:

- ❑ Timer (HTIMER) input—The timer to be stopped. You should stop a DosStartTimer when you are done with it, so that OS/2 does not continue posting it after every time interval.

```
/*  prog13f5.   DoStartTimer.                                      */
/*                                                                 */

#define INCL_DOS
#include <os2.h>

int main (int argc, void *argv[])
{
   /* This program sounds an alarm every 5 seconds,                */
   /* for 30 seconds.                                              */

   HEV    ESem;
   HTIMER Timer;
   ULONG  AlarmTime;
   int    i;
   ULONG  j;
   int    rc;

   /* Set time in milliseconds                                     */
   AlarmTime  = 5000;

   /* Create a shared, unnamed event semaphore                     */
   DosCreateEventSem (NULL,                  /* unnamed            */
                      &ESem,                 /* ptr to handle      */
                      DC_SEM_SHARED,         /* shared             */
                      0);                    /* initially 'set'    */

   rc = DosStartTimer(AlarmTime, (HSEM) ESem, &Timer);
   if (rc) printf("DosStartTimer failed.  RC = %ld\n", rc);

   j = 0;
   /* In this loop, we increment the number of alarms              */
   /* sounded not by 1 but by the number of times                  */
   /* the semaphore was posted.  This is in case 10                */
   /* seconds or more passes before the program is                 */
   /* scheduled to run again.                                      */

   for (i=0; i<6 ; i++ ) {
      DosWaitEventSem(ESem, SEM_INDEFINITE_WAIT);
      DosResetEventSem(ESem, &j);            /* Find # times posted */
      for (; j>0 ; j--) {
        DosBeep(1300, 0500);
      }                                                    /* endfor */
    }                                                      /* endfor */

    DosStopTimer(Timer);
    return 0;
}
```

Figure 13-5. *The DosStartTimer API.*

SUMMARY

This chapter discusses two different concepts of time—real time and CPU time—and shows that OS/2 2.1 provides you with the ability to work with real time or relative time, whichever is better suited to your application. It goes into some detail about how a computer simulates a real world clock using electric pulses and counters, and how to retrieve one of those counters directly. This chapter fully explains and illustrates all of the explicitly timer-related APIs in OS/2 2.1, namely:

- ❑ DosSetDateTime—Sets the system's date and time.
- ❑ DosGetDateTime—Returns the current value of the system's date and time.
- ❑ DosSleep—Suspends a process or thread for a specified amount of CPU time.
- ❑ DosAsynchTimer—Creates a timer which will post a semaphore when a specified amount of real time has elapsed. The caller often suspends itself then by waiting on the semaphore.
- ❑ DosStartTimer—Creates a repeated timer that will post a semaphore every time a specified amount of real time has elapsed, until the timer is stopped. The caller often suspends itself by waiting on the semaphore.
- ❑ DosStopTimer—Destroys a timer created with the DosAsynchTimer API or the DosStartTimer API and wakes up any processes waiting on that timer.

Some of the occasions when timers are useful are the following:

- ❑ When your program interacts with a user.
- ❑ When your program works with a real time device or application.
- ❑ When you want to have tighter control over yielding resources.
- ❑ When you have a multithreaded or multiprocess application.

Finally, this chapter points out that OS/2 2.1 has APIs that use implicit timers, such as some of those associated with semaphores and pipes to allow you to specify how long you are willing to wait for something to happen.

CHAPTER 14

Error, Exception, and Message Management

"Every great mistake has a halfway moment, a split second when it can be recalled and perhaps remedied."
—Pearl Buck

INTRODUCTION

Errors will always occur. Error management is the planning for error conditions. By managing error conditions for the most common error events your program may have, you can make your program more usable.

OS/2 provides some assistance in managing OS/2 API errors. When an OS/2 Control Program API returns with a code other than 0, this code indicates why the API failed. Within your program, you may be able to do something that can correct the situation when certain codes are returned.

This chapter will discuss errors and provide some assistance in how to manage errors that occur during the execution of your program.

API ERROR MANAGEMENT

When a Control Program API fails, it returns an error code to the caller. The error code equates to a *preprocessor definition* found in the bseerr.h file. These start with *ERROR_*. Error definitions can be referenced in your program by first defining INCL_DOSERRORS and before including the os2.h file. The preprocessor definition name for the error indicates what the error is. For example, the DosDeleteDir API returns a code of 3 (ERROR_PATH_NOT_FOUND) when a directory name with an invalid path is passed.

Part of a program's design should consider what to do when an API fails. Your application may be able to take corrective action after examining the return code. If the application cannot, then it may be appropriate to display some information about the error to the user or log it to a file. OS/2 2.1 provides the DosErrClass API which returns a classification, course of action, and error originator for a particular error code.

DosErrClass

```
DosErrClass(
            ULONG code,
            PULONG pClass,
            PULONG pAction,
            PULONG pLocus);
```

Parameters:

- ❑ code (ULONG) input—A non-zero return code from a Control Program API.
- ❑ pClass (PULONG) output—A buffer pointer where the error classification code is returned.

Name	Value	Description
ERRCLASS_OUTRES	(1)	Out of resources.
ERRCLASS_TEMPSIT	(2)	Temporary situation.
ERRCLASS_AUTH	(3)	Authorization.
ERRCLASS_INTRN	(4)	Internal error.
ERRCLASS_HRDFAIL	(5)	Hardware failure.
ERRCLASS_SYSFAIL	(6)	System failure.
ERRCLASS_APPERR	(7)	Application error.
ERRCLASS_NOTFND	(8)	Object not found.
ERRCLASS_BADFMT	(9)	Bad format.
ERRCLASS_LOCKED	(10)	Locked.
ERRCLASS_MEDIA	(11)	Media failure, cyclic redundancy check (CRC) error.
ERRCLASS_ALREADY	(12)	Action already done.
ERRCLASS_UNK	(13)	No classification.
ERRCLASS_CANT	(14)	Action can not be performed.
ERRCLASS_TIME	(15)	Timeout.

- ❑ pAction (PULONG) output—A buffer pointer where a code that illustrates the corrective action is returned.

Name	Value	Description
ERRACT_RETRY	(1)	Retry.
ERRACT_DLYRET	(2)	Retry after pausing.
ERRACT_USER	(3)	The user input was bad; therefore ask for it again.
ERRACT_ABORT	(4)	Abort but allow program clean-up.
ERRACT_PANIC	(5)	Abort immediately.
ERRACT_IGNORE	(6)	Ignore the error.
ERRACT_INTRET	(7)	Retry after the user has intervened.

- ❑ pLocus (PULONG) output—A buffer pointer where a code that indicates the error originator is returned.

Name	Value	Description
ERRLOC_UNK	(1)	Unknown originator
ERRLOC_DISK	(2)	Mass storage device such as a disk
ERRLOC_NET	(3)	Network
ERRLOC_SERDEV	(4)	Serial device
ERRLOC_MEM	(5)	Memory

The primary use for the DosErrClass API is to help facilitate a corrective action. This API is also useful to help document errors that the application can't correct or didn't expect. The program in Figure 14-1 illustrates the use of this API to log unexpected error information about the program.

```
/*                                                                  */
/* prog14f1.c.  Logging unexpected error information                */
/*                                                                  */

#define INCL_DOSMISC
#define INCL_DOSDATETIME
#include <os2.h>

#include <stdio.h>

VOID LogError(APIRET APIrc, ULONG ulSourceLine,
              PSZ pszSourceFile)
{
   FILE * logfile;
   ULONG Class, Action, Locus;
   DATETIME dt;

   if ((logfile = fopen("ERROR.LOG", "a")) == NULL) {
      printf("Couldn't open log file ERROR.LOG\n");
      return;
   }
   DosErrClass(APIrc, &Class, &Action, &Locus);
   DosGetDateTime(&dt);
   fprintf(logfile, "Code Module %s, line %ld ",
           pszSourceFile, ulSourceLine);
   fprintf(logfile, "logged error %ld ", APIrc);
   fprintf(logfile, "with error class = %ld, ", Class);
   fprintf(logfile, "error action = %ld, ", Action);
   fprintf(logfile, "and originator = %ld ", Locus);
   fprintf(logfile, "on date %d/%d/%d ", dt.month,
           dt.day, dt.year - 1900);
   fprintf(logfile, "at time %d:%d:%d.\n", dt.hours,
           dt.minutes, dt.seconds);
   fclose(logfile);
}

INT main(VOID)
{
   APIRET APIrc;

   APIrc = DosDeleteDir("THISDIR");
   if (APIrc != 0) LogError(APIrc, __LINE__, __FILE__);

   APIrc = DosDelete("THISFILE.TXT");
   if (APIrc != 0) LogError(APIrc, __LINE__, __FILE__);

   return(0L);
}
```

Figure 14-1. *Program using DosErrClass in an error logging facility.*

The program in Figure 14-1 implements an error logging facility. This facility is used for writing an error string to a log file which provides information about the error. The LogError function writes one line to an error file each time it is called. For simplicity, the C standard library file functions are used to write the information to a file. The error line contains the current date and time, the program module name, the line number within the module, and the error code along with its associated DosErrClass information. The date and time are retrieved using the DosGetDateTime API. The module name and line number are obtained using the C predefined macros __LINE__ and __FILE__.

Given that the program is stored in a module named LOG.C, Figure 14-2 shows the information written to file ERROR.LOG when the program is run.

```
Code Module LOG.C, line 39 logged error 3 with error class = 8, error
action = 3, and originator = 2 on date 5/28/93 at time 14:27:34.
Code Module LOG.C, line 42 logged error 2 with error class = 8, error
action = 3, and originator = 2 on date 5/28/93 at  time 14:27:34.
```

Figure 14-2. *The information written to file ERROR.LOG by PROG14F1.*

Notice that the two errors logged in Figure 14-2 both have the error class of ERRCLASS_NOTFND (8), error action of ERRACT_USER (3), and originator of ERRLOC_DISK (2). This indicates that an act of searching on the disk resulted in a not found condition because of the user input (API parameter).

OS/2 2.1 provides a built in mechanism to inform the user about hard errors and exceptions. Hard errors are those where hardware such as a disk drive reports the error and exceptions are unexpected program errors. When one of these errors occurs, OS/2 either aborts the program and provides information about the reason for the abort or informs the user in case the user can correct the situation.

An abort will occur automatically when a program error happens, such as attempting to reference memory that is not owned. In this event, the user has no way of fixing the program while it is running. Therefore, OS/2 ends the program and displays information about the abort, such as why the program was aborted along with the contents of the processor's registers.

A typical case where OS/2 prompts the user automatically because of an error occurs when the floppy disk drive has no disk inserted. If the program attempts to read from the A: drive but there is no disk present, the user will be told this and asked for a response. This is a convenient feature for a typical error that the operating system provides.

However, certain programs, or at certain times any program, may not want these popups to occur. For instance, your program might want to display this information itself. OS/2 always displays this information in the operating system's national language.

If your program must run in many different languages, on any language version of OS/2, it would be more appropriate to display these errors in the program's language. The DosError API is used for disabling and enabling these error popups during a program's execution.

DosError

```
          DosError(
                  ULONG   error);
```

Parameter:

- ❑ error (ULONG) input—A bit field that specifies how to manage hard error and program exception pop-ups.

Name	Value	Description
FERR_DISABLEHARDERR	(0x00000000L)	Disable pop-ups for hard errors.
FERR_ENABLEHARDERR	(0x00000001L)	Enable pop-ups for hard errors.
FERR_ENABLEEXCEPTION	(0x00000000L)	Enable popups for program exceptions.
FERR_DISABLEEXCEPTION	(0x00000002L)	Disable popups for program exceptions.

EXCEPTION MANAGEMENT

During program execution, unexpected errors can occur that will cause the program to terminate. This is the default behavior that occurs with operations such as dividing by 0 and referencing memory that your process doesn't own. OS/2 calls this abnormal condition an exception and includes other conditions such as I/O errors and user interrupts (Ctrl+Break) within this definition.

Exceptions can occur with the code executed on the current thread or they can occur asynchronously due to events such as the user pressing Ctrl+Break. There are three categories of exceptions: *signal* exceptions, *system* exceptions, and *user-defined* exceptions.

Signal Exceptions

Signal exceptions will occur when the end user presses either the Ctrl+Break (XCPT_SIGNAL_BREAK) or the Ctrl+C (XCPT_SIGNAL_INTR) in programs except those running under Presentation Manager. A program can send these signal exceptions using the DosSendSignalException API.

DosSendSignalException

```
DosSendSignalException(
                                PID    pid,
                                ULONG  exception);
```

Parameters:

- ❑ pid (PID) input—The process identifier for the process that is to receive the signal.

- ❑ exception (ULONG) input—The signal exception type. Valid types are:

```
XCPT_SIGNAL_INTR  (1)   A Ctrl-C signal
XCPT_SIGNAL_BREAK (4)   A Ctrl-Break signal
```

A signal also occurs when a DosKillProcess (XCPT_SIGNAL_KILLPROC) call is made. The default processing of these three signals is to exit the process. If your process wants to react to these signals, it must register an exception handler using the DosSetExceptionHandler API.

DosSetExceptionHandler

```
DosSetExceptionHandler(
                        PEXCEPTIONREGISTRATIONRECORD pERegRec);
```

Parameter:

- ❑ pERegRec (PEXCEPTIONREGISTRATIONRECORD) input—A pointer to an EXCEPTIONREGISTRATIONRECORD that contains the address of the exception handler routine that is to be registered.

Exception handlers are registered with the thread and not the process. To register an exception handler to manage the signal exceptions, you must make the registration call on the main thread (thread 1). After an exception handler to manage signal exceptions is registered, the process should call the DosSetSignalExceptionFocus API on the same thread so that the exception handler can receive signal exceptions. Since a program can have child processes, it is necessary to indicate which process should have signal focus.

DosSetSignalExceptionFocus

```
DosSetSignalExceptionFocus(
                            BOOL32  flag,
                            PULONG  pulTimes);
```

Parameters:

- ❑ flag (BOOL32) input—A flag field indicating whether to set or remove signal exception focus for the process.

```
SIG_UNSETFOCUS (0)   Stop receiving signals
SIG_SETFOCUS   (1)   Start receiving signals
```

- ❑ pulTimes (PULONG) output—Pointer to a buffer where the number of times the API has been called with SIG_SETFOCUS minus the number of times called with SIG_UNSETFOCUS on the current process is returned.

When an exception handler is no longer needed for the thread it was registered on, the DosUnsetExceptionHandler API should be called to remove the handler.

DosUnsetExceptionHandler

```
DosUnsetExceptionHandler(
                          PEXCEPTIONREGISTRATIONRECORD pERegRec);
```

Parameter:

- ❑ pERegRec (PEXCEPTIONREGISTRATIONRECORD) input—A pointer to an EXCEPTIONREGISTRATIONRECORD that contains the address of the exception handler routine that is to be unset.

The program in Figure 14-3 registers an exception handler to monitor for signal exceptions. Each time the user presses Ctrl+C, causing signal XCPT_SIGNAL_INTR to occur, the exception handler displays a message for you and returns XCPT_CONTINUE_EXECUTION indicating that the exception has been processed. When the user has pressed Ctrl+C three times, the program terminates.

```
/*                                                                    */
/* prog14f3.  Register exception handler to manage signal exception */

#define INCL_DOSEXCEPTIONS
#define INCL_DOSERRORS
#define INCL_DOSSEMAPHORES
#include <os2.h>

HEV mysem;
```

```
SHORT count = 0;

ULONG _System MySignalHandler(PEXCEPTIONREPORTRECORD,
                PEXCEPTIONREGISTRATIONRECORD,
                PCONTEXTRECORD, PVOID);

ULONG _System MySignalHandler(PEXCEPTIONREPORTRECORD pRepRec,
                PEXCEPTIONREGISTRATIONRECORD pRegRec,
                PCONTEXTRECORD pConRec, PVOID pDispCon)
{
   if (pRepRec->ExceptionNum == XCPT_SIGNAL) {
      if (pRepRec->ExceptionInfo[0] == XCPT_SIGNAL_INTR) {
         printf("Ctrl+C occured!!!\n");
         if (count == 2) {
            DosPostEventSem(mysem);
         }
         count++;
         return(XCPT_CONTINUE_EXECUTION);
      } else {
         return(XCPT_CONTINUE_SEARCH);
      }
   } else {
      return(XCPT_CONTINUE_SEARCH);
   }
}

INT main(VOID)
{
   EXCEPTIONREGISTRATIONRECORD ERegRec;
   ULONG count;

   DosCreateEventSem((PSZ)NULL, &mysem, 0, FALSE);
   ERegRec.prev_structure = NULL;
   ERegRec.ExceptionHandler = MySignalHandler;
   DosSetExceptionHandler(&ERegRec);
   DosSetSignalExceptionFocus(SIG_SETFOCUS, &count);

   while (DosWaitEventSem(mysem, SEM_INDEFINITE_WAIT) ==
          ERROR_INTERRUPT);

   DosSetSignalExceptionFocus(SIG_UNSETFOCUS, &count);
   DosUnsetExceptionHandler(&ERegRec);
   DosCloseEventSem(mysem);

   return(0);
}
```

Figure 14-3. *Program illustrating the registration of an exception handler to manage signal exceptions.*

The program in Figure 14-3 waits on an event semaphore which is posted when Ctrl+C is pressed the third time. This will cause the DosWaitEventSem API to return successfully, which allows the program to complete. You will notice that the DosWaitEventSem API is called within the body of a while loop that only breaks out when the API return code is something other than ERROR_INTERRUPT.

The ERROR_INTERRUPT code is returned whenever a thread is waiting on a semaphore and an exception occurs. When an exception occurs, the exception handler is executed on the thread in which the exception occurred. In order to process an exception, a thread waiting on a semaphore must be released from the wait state. For this program, it was suitable to simply call the DosWaitEventSem again when an exception occurred.

The exception handler, MySignalHandler, in Figure 14-3 has four parameters. The first parameter is a pointer to an EXCEPTIONREPORTRECORD. This structure provides information about what the exception is and specific information about the exception.

EXCEPTIONREPORTRECORD

```
struct _EXCEPTIONREPORTRECORD {
     ULONG   ExceptionNum;
     ULONG   fHandlerFlags;
     struct  _EXCEPTIONREPORTRECORD
     *NestedExceptionReportRecord;
     PVOID   ExceptionAddress;
     ULONG   cParameters;
     ULONG   ExceptionInfo[EXCEPTION_MAXIMUM_PARAMETERS];
     };
```

Parameters:

- ❑ ExceptionNum (ULONG) - The exception number.

- ❑ fHandlerFlags (ULONG) - Exception attribute flags.

```
EH_NONCONTINUABLE (0x1)   Non-continuable exception
EH_UNWINDING      (0x2)   Unwinding in progress
EH_EXIT_UNWIND    (0x4)   Full unwinding in progress
EH_STACK_INVALID  (0x8)   Debuggee stack invalid
EH_NESTED_CALL    (0x10)  Nested exception
```

- ❑ NestedExceptionReportRecord (PEXCEPTIONREPORTRECORD)—When an exception is nested, this is a pointer to an execption report record .

- ExceptionAddress (PVOID)—XCPT_DATA_UNKNOWN or the address where the exception occurred.
- cParameters (ULONG)—The number of entries used in the ExceptionInfo array.
- ExceptionInfo (ULONG [])—An array which contains cParameters number of entries specific to the exception.

The second parameter to an exception handler is a pointer to a structure, EXCEPTIONREGISTRATIONRECORD. This structure contains two fields, one containing the exception handler's function address and the other containing the address to the EXCEPTIONREGISTRATIONRECORD for the exception handler that was installed previously. This same structure is used to set an exception handler using the DosSetExceptionHandler API.

The third parameter to an exception handler is a pointer to a CONTEXTRECORD structure. This structure contains the contents of the processor registers at the time the exception occurred. The first field in this structure, ContextFlags, is used to identify what registers can be referenced for the particular exception. The possible settings for ContextFlags are:

```
CONTEXT_CONTROL        (0x00000001L)

With this flag set, the ctx_RegEbp, ctx_RegEip, ctx_SegCs, ctx_EFlags,
ctx_RegEsp,  and ctx_SegSs structure fields can be referenced.

CONTEXT_INTEGER        (0x00000002L)

With this flag set,  the ctx_RegEdi, ctx_RegEsi, ctx_RegEax,
ctx_RegEbx, ctx_RegEcx, and ctx_RegEdx structure fields can be
referenced.

CONTEXT_SEGMENTS       (0x00000004L)

With this flag set, the ctx_SegGs, ctx_SegFs, ctx_SegEs, and ctx_SegDs
structure fields can be referenced.

CONTEXT_FLOATING_POINT (0x00000008L)

With this flag set, the ctx_env and ctx_stack structure fields can be
referenced.
```

The fourth parameter to an exception handler is a pointer to a reserved field.

If an exception handler processes an exception, the handler should return a value in XCPT_CONTINUE_EXECUTION to indicate that the exception has been handled. By returning this value during the process of handling a signal exception, the signal is acknowledged.

An exception handler can also use the DosAcknowledgeSignalException API to acknowledge a signal.

DosAcknowledgeSignalException

```
DosAcknowledgeSignalException(
                                ULONG ulSignalNum);
```

Parameter:

- ❑ ulSignalNum (ULONG) input—The signal number to be acknowledged. The available signals are:

```
XCPT_SIGNAL_INTR     (1)  Acknowledges the Ctrl+C  signal.
XCPT_SIGNAL_KILLPROC (3)  Acknowledges the signal that occurs
                            when another process calls
                            theDosKillProcess API.
XCPT_SIGNAL_BREAK    (4)  Acknowledges the Ctrl+Break signal.
```

If the exception has not been handled, the exception handler should return XCPT_CONTINUE_SEARCH, which indicates to pass the exception on to the previously registered exception handler. The DosSetExceptionHandler API can be called multiple times on a thread and can thereby nest exception handlers. The last exception handler registered will receive an exception first. Because of this, the EXCEPTIONREGISTRATIONRECORD must be on the stack. A pointer to this structure is passed to the DosSetExceptionHandler and DosUnsetExceptionHandler APIs. Before the function that issued the DosSetExceptionHandler API returns, the DosUnsetExceptionHandler API must be called.

During certain circumstances, while running your program, you may find it inappropriate to have an asynchronous exception, such as a signal exception, occur. OS/2 provides a mechanism whereby you can defer asynchronous exceptions. It is important to use this mechanism around critical code to ensure that resources are properly freed before the thread is terminated. The DosEnterMustComplete and DosExitMustComplete APIs should be called around the code that must not be interrupted by an asynchronous exception.

DosEnterMustComplete

```
        DosEnterMustComplete(
                               PULONG pulNesting);
```

Parameter:

- ❑ pulNesting (PULONG) output—A pointer to a buffer where the number of calls to the DosEnterMustComplete API minus the number of calls to the DosExitMustComplete API for the calling thread is returned.

DosExitMustComplete

```
        DosExitMustComplete(
                              PULONG pulNesting);
```

Parameter:

- ❑ pulNesting (PULONG) output—A pointer to a buffer where the number of calls to the DosEnterMustComplete API minus the number of calls to the DosExitMustComplete API for the calling thread is returned.

System Exceptions

The operating system provides a class of exceptions, called system exceptions, that occur during certain events. When these events occur, the default action in most cases is to terminate the thread. Signal exceptions are included in the class of system exceptions. Synchronous exceptions such as protection violations are also included. Figure 14-4 contains the full list of signal exceptions and their values.

```
XCPT_ACCESS_VIOLATION          (0xC0000005)
XCPT_ARRAY_BOUNDS_EXCEEDED     (0xC0000093)
XCPT_ASYNC_PROCESS_TERMINATE   (0xC0010002)
XCPT_B1NPX_ERRATA_02           (0xC0010004)
XCPT_BAD_STACK                 (0xC0000027)
XCPT_BREAKPOINT                (0xC000009F)
XCPT_DATATYPE_MISALIGNMENT     (0xC000009E)
```

XCPT_FLOAT_DENORMAL_OPERAND	(0xC0000094)
XCPT_FLOAT_DIVIDE_BY_ZERO	(0xC0000095)
XCPT_FLOAT_INEXACT_RESULT	(0xC0000096)
XCPT_FLOAT_INVALID_OPERATION	(0xC0000097)
XCPT_FLOAT_OVERFLOW	(0xC0000098)
XCPT_FLOAT_STACK_CHECK	(0xC0000099)
XCPT_FLOAT_UNDERFLOW	(0xC000009A)
XCPT_GUARD_PAGE_VIOLATION	(0x80000001)
XCPT_ILLEGAL_INSTRUCTION	(0xC000001C)
XCPT_INTEGER_DIVIDE_BY_ZERO	(0xC000009B)
XCPT_INTEGER_OVERFLOW	(0xC000009C)
XCPT_INVALID_DISPOSITION	(0xC0000025)
XCPT_INVALID_LOCK_SEQUENCE	(0xC000001D)
XCPT_INVALID_UNWIND_TARGET	(0xC0000028)
XCPT_IN_PAGE_ERROR	(0xC0000006)
XCPT_NONCONTINUABLE_EXCEPTION	(0xC0000024)
XCPT_PRIVILEGED_INSTRUCTION	(0xC000009D)
XCPT_PROCESS_TERMINATE	(0xC0010001)
XCPT_SIGNAL	(0xC0010003)
XCPT_SINGLE_STEP	(0xC00000A0)
XCPT_UNABLE_TO_GROW_STACK	(0x80010001)
XCPT_UNWIND	(0xC0000026)

Figure 14-4. *System exceptions.*

In the OS/2 2.1 Toolkit, the bsexcpt.h file describes these system exceptions along with the parameters that arrive with the exception.

Using the DosAllocMem API, your program can allocate memory, but it doesn't have to commit the memory at the same time. When another 4K page is needed, the DosSetMem API can be called to commit the page. If an attempt is made in your program to access a page in a memory object that has not been committed, an exception will occur. The information provided by the exception allows you to commit the page during the processing of the exception. The program in Figure 14-5 supports the system exception that occurs when the program accesses a memory object page that has not been committed.

```
/*                                                                */
/* prog14f5.  Exceptionhandler to manage uncommitted mem obj access */
/*                                                                */

#define INCL_DOSMEMMGR
#define INCL_DOSEXCEPTIONS
#include <os2.h>

#include <stdlib.h>

ULONG _System MyMemoryHandler(PEXCEPTIONREPORTRECORD,
                PEXCEPTIONREGISTRATIONRECORD,
                PCONTEXTRECORD, PVOID);

ULONG _System MyMemoryHandler(PEXCEPTIONREPORTRECORD pRepRec,
                PEXCEPTIONREGISTRATIONRECORD pRegRec,
                PCONTEXTRECORD pConRec, PVOID pDispCon)
{
   ULONG ulSize, ulAttributes;
   APIRET rc;

   /* Check for Read/Write access violations that pass              */
   /* a known memory address.                                       */
   if ((pRepRec->ExceptionNum == XCPT_ACCESS_VIOLATION) &&
       (pRepRec->ExceptionAddress !=
                               (PVOID)XCPT_DATA_UNKNOWN)) {
      if (((pRepRec->ExceptionInfo[0] == XCPT_READ_ACCESS) ||
           (pRepRec->ExceptionInfo[0] == XCPT_WRITE_ACCESS))
          && (pRepRec->ExceptionInfo[1] !=
                                       XCPT_DATA_UNKNOWN)) {

         /* Find out if the memory just needs to be                 */
         /* committed by checking the attributes.                   */
         ulSize = 1;
         rc = DosQueryMem((PVOID)pRepRec->ExceptionInfo[1],
                     &ulSize, &ulAttributes);
         if (!rc &&
             !(ulAttributes & (PAG_COMMIT | PAG_FREE))) {

            /* Commit the memory and return indicating              */
            /* the exception has been processed and the             */
            /* program can continue execution.                      */
            rc = DosSetMem((PVOID)pRepRec->ExceptionInfo[1],
                           0x1000, PAG_COMMIT | PAG_DEFAULT);
            if (!rc) {
               printf("Page committed!!!\n");
               return(XCPT_CONTINUE_EXECUTION);
            }
         }
      }
   }
   return(XCPT_CONTINUE_SEARCH);
}

INT main(VOID)
{
   EXCEPTIONREGISTRATIONRECORD ERegRec;
   PVOID memobj;
   ULONG count;

   ERegRec.prev_structure = NULL;
   ERegRec.ExceptionHandler = MyMemoryHandler;
   DosSetExceptionHandler(&ERegRec);

   DosAllocMem(&memobj, 6000, PAG_READ|PAG_WRITE);
   memset(memobj, '\0', 6000);
   DosFreeMem(memobj);

   DosUnsetExceptionHandler(&ERegRec);

   return(0);
}
```

***Figure 14-5**. Program illustrating the registration of an exception handler to manage uncommitted memory object access violations.*

The IBM C Set ++ product features three notable library functions that are helpful in easily handling certain system exceptions. The library functions *signal, setjmp,* and *longjmp* are a convenient method for supporting basic exception protection. The program in Figure 14-6 represents a typical use for this type of protection scheme.

```
/*                                                                      */
/* prog14f6.  Functions available with IBM C Set++                      */
/*                                                                      */

#include <os2.h>
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

#define CODE_PSZ 1
#define CODE_NUM 2
INT ExternalFunction(PVOID, USHORT);

INT main(VOID)
{
   INT iRC;

   iRC = ExternalFunction("mystr", CODE_PSZ);
   iRC = ExternalFunction((PVOID)1, CODE_NUM);
   iRC = ExternalFunction((PVOID)1, CODE_PSZ);

   return(OL);
}

static jmp_buf SigBuffer;

VOID SEGV_Handler(int sig)
{
   longjmp(SigBuffer, -1);
}
 INT ExternalFunction(PVOID p, USHORT usCode)
{
   PVOID pPrevExceptionHandler;
   INT iRC;
   /* Register signal handler                                    */
   pPrevExceptionHandler = signal(SIGSEGV, SEGV_Handler);
   if (setjmp(SigBuffer) != 0) }
      iRC = 1;                                /* Bad parameter */
      printf("Signal occurred!!!\n");
      goto errorexit;
   {

   /*  Access pointers passed in                                 */
   switch (usCode)}
      case CODE_PSZ;
         printf("This string is %s with length %d\n",
            (PSZ)p, strlen((PSZ)p));
         break;  }
      default:   ;
   {

   /* Deregister signal handler                                  */
   signal(SIGEGV, pPrevExceptionHandler);
   iRC = 0;

   errorexit:

   return(iRC);
}
```

Figure 14-6. *Program demonstrating the signal, setjmp, and longjmp functions available for exception condition management with IBM C Set ++.*

The program in Figure 14-6 protects against segmentation violations that can occur when you reference the parameter **p**, which is passed in to the function ExternalFunction. You will notice in the main function that the third call to this function passes CODE_PSZ instead of CODE_NUM, which is a coding error. Without the protection code that exists around the access of this pointer, a segmentation violation would happen within the ExternalFunction routine. The segmentation violation will occur when you call strlen on the **p** parameter. Instead, the function falls out with an error condition to the caller.

The ExternalFunction, before accessing the **p** variable, registers a signal handler for SIGSEGV signals. This registration means that when a SIGSEGV occurs, the function whose address is passed in the second parameter to the signal function will be called to notify it about the signal. Therefore, function SEGV_Handler will be called when this occurs. After the signal is registered, the setjmp function is called to preserve the state of the thread such that if the exception occurs, it can recover and jump over the code that caused the exception. Within the SEGV_Handler signal routine, the longjmp function is called, which causes the program to jump back to where the setjump call was issued.

When a signal occurs, the signal handler is automatically deregistered. If no signal occurs, the program should deregister the signal handler before falling out so that it doesn't get called erroneously. Signal handlers can be nested. Therefore, it is important to deregister using the address of the previously installed handler function.

Handling exceptions in this manner is more usable than letting the program terminate because of this type of error. In some cases, even though an error this severe occurs, the program can still continue.

User-Defined Exceptions

The exception management interface also allows you to create your own exceptions that can be managed within your exception handler. These user-defined exceptions should have an exception number that is unique and different from the system exceptions listed in Figure 14-4.

An exception handler should be registered for the thread. When the exception occurs, an EXCEPTIONREPORTRECORD should be filled in with the information about the user-defined exception. To invoke the exception handler to manage an exception, isssue a DosRaiseException API, on which you pass a pointer to the structure, the EXCEPTIONREPORTRECORD.

DosRaiseException

```
DosRaiseException(
                        PEXCEPTIONREPORTRECORD pexcept);
```

Parameter:

❑ pexcept (PEXCEPTIONREPORTRECORD) input—A pointer to an EXCEPTIONREPORTRECORD that contains the information about the exception that is to be raised.

MESSAGE MANAGEMENT

When an error occurs, it may be necessary to display a message about the error to the user. OS/2 provides a facility for creating a file that contains messages that can be extracted by a program during execution. This is important for program national language enablement as well. The MKMSGF tool is used to build the message file. This tool takes as input a number of parameters and a message definition file. An example of a message definition file is shown in Figure 14-6.

```
; This is the message input file for component CMP
CMP
CMP0200E: A file name must be specified.
CMP0201?:
CMP0202H: Syntax: MKMSGF infile[.ext] outfile[.ext]
CMP0203?:
CMP0204I: All information stored
CMP0205W: Warning!  All files will be deleted.
CMP0206?:
CMP0207P: Enter the name of the file: %0
CMP0208E: The %1 parameter is not valid.
```

***Figure 14-6**. Message definition file example.*

Message definition files can include comments, a component identifier, and component messages. Comment lines must start with a semicolon in column 1 and can contain any information. The component identifier is used to indicate the message prefix that should be used. Each component message begins with a message header, which consists of the first 10 characters on the line arranged in the following order:

- ❑ A component identifier. In Figure 14-6, this is CMP.
- ❑ A four-digit number. The first message number can be any value but each subsequent component message number must be one higher.
- ❑ A message type identifier. This can be either E for Error, ? for no message assigned, H for Help, W for Warning, I for Information, or P for Prompt.
- ❑ A colon and a blank character.

Following the message header is the message text. To support string substitution at run time, %1 through %9 can be put within the body of the text. The message retrieval

API offers the ability to substitute up to nine strings in place of the %1 through %9 entries within the message. To indicate to the message retrieval API not to put a carriage return and line feed character at the end of the message, you can place a %0 at the end of the message. This allows for the program to prompt the user for input information on the same line as the message.

The syntax for creating a message file with the MKMSGF utility is the following:

```
MKMSGF infile outfile [options]
```

or

```
MKMSGF @controlfile
```

The options available are:

- ❑ /?—Display the MKMSGF syntax.
- ❑ /V—MKMSGF shows information pertaining to the building of the message file.
- ❑ /D—With this option you can set the DBCS language and country identifier in the format /Ddbcslang/countryid.
- ❑ /P—With this option you can specify the code page in the format /Pcodepage.
- ❑ /L—With this option you can set the language family ID and the language version ID in the format /Lfamily,version.

MKMSGF also allows a control file to be passed when a message file needs to be built that contains message definitions for multiple code pages. On each line of the control file, the same syntax is used. It consists of specifying the input file, the output file, and the MKMSGF options.

The messages found within the output file that MKMSGF generates can be loaded at run time using the message retrieval API. However, some applications might want to bind the messages to their executable (.EXE) file. Binding the messages to the executable makes extracting your messages faster, and one less file is needed to run your program. The MSGBIND utility is used to bind message files to an executable file.

The MSGBIND tool takes one parameter, a control file that contains information about what is to be bound and where. The control file contains the name of the executable, the name of the message file, and the message numbers from the message file that are to be bound.

Figure 14-7 is an example input file that binds 6 messages from the myprog.msg file to the myprog.exe file.

```
>c:\myprog.exe
<c:\myprog.msg
CMP0200
CMP0202
CMP0204
CMP0205
CMP0207
CMP0208
```

Figure 14-7. *Example MSGBIND input file.*

The greater than symbol in column 1 followed by a file name is used to denote which executable the messages should be bound to. The less than symbol in column 1 followed by a file name is used to denote the message file that the subsequent messages should be taken from and bound to the executable. The MSGBIND control file can specify multiple executable and message files.

Once the messages are either bound to an executable or stored in a message file, they are ready to be retrieved during execution by using the message retrieval API DosGetMessage. The DosGetMessage API, when searching for the requesting message, will check the executable first for a bound message and, if not found, it will search the message file.

DosGetMessage

```
DosGetMessage(
             PCHAR* pTable,
             ULONG  cTable,
             PCHAR  pBuf,
             ULONG  cbBuf,
             ULONG  msgnumber,
             PSZ    pszFile,
             PULONG pcbMsg);
```

Parameters:

- ❑ pTable (PCHAR *) input—A pointer to an array of strings to be used for text substitution with the message that is retrieved.
- ❑ cTable (ULONG) input—The number of elements in the pTable array.
- ❑ pBuf (PCHAR) output—A pointer to an output buffer where the message can be written to.
- ❑ cbBuf (ULONG) input—The size in bytes of the pBuf buffer.
- ❑ msgnumber (ULONG) input—The message number that is associated with the message that is to be retrieved.
- ❑ pszFile (PSZ) input—The file name where the message can be found.
- ❑ pcbMsg (PULONG) output—The number of bytes the message occupies in the buffer pBuf.

The DosGetMessage API also provides the ability to do string substitution and replace the %1 through %9 fields found within the base message with strings provided through the pTable array. The DosInsertMessage API provides string substitution for base strings that don't have to be first loaded from a message file.

DosInsertMessage

```
DosInsertMessage(
                   PCHAR* pTable,
                   ULONG  cTable,
                   PSZ    pszMsg,
                   ULONG  cbMsg,
                   PCHAR  pBuf,
                   ULONG  cbBuf,
                   PULONG pcbMsg);
```

Parameters:

- ❑ pTable (PCHAR *) input—A pointer to an array of strings to be used for text substitution with the message that is retrieved.

- ❑ cTable (ULONG) input—The number of elements in the pTable array.
- ❑ pszMsg (PSZ) input—The base string used for substitution.
- ❑ cbMsg (PSZ) input—The number of bytes of the base string in pszMsg.
- ❑ pBuf (PCHAR) output—A pointer to an output buffer where the message can be written to.
- ❑ cbBuf (ULONG) input—The size in bytes of the pBuf buffer.
- ❑ pcbMsg (PULONG) output—The number of bytes the message occupies in the buffer pBuf.

The MKMSGF tool provides the capability to store messages under multiple code pages. The DosQueryMessageCP API is used to query the code pages and language identifiers that are used with a message file. This is useful information in the event you may need to change code pages during runtime to support the messages on a particular code page installation of OS/2.

The DosQueryMessageCP

```
DosQueryMessageCP(
                          PCHAR  pb,
                          ULONG  cb,
                          PSZ    pszFilename,
                          PULONG cbBuf);
```

Parameters:

- ❑ pb (PCHAR) output—A pointer to an output buffer where the list of code pages and language identifiers associated with the message file is returned.
- ❑ cb (ULONG) input—The size in bytes of the buffer pointed to by the pb parameter.
- ❑ pszFilename (PSZ) input—The message file name.
- ❑ cbBuf (PULONG) output—The number of bytes that were written into the pb buffer.

OS/2 2.1 also provides a message API to write a message to a file or device. This DosPutMessage API has a simple interface for doing this.

DosPutMessage

```
          DosPutMessage(
                        HFILE hfile,
                        ULONG cbMsg,
                        PCHAR pBuf);
```

Parameters:

- ❑ hFile (HFILE) input—The handle to the device or file where the message is to be written to.
- ❑ cbMsg (ULONG) input—The number of bytes that are to be written.
- ❑ pBuf (PCHAR) input—A pointer to the buffer containing the message that is to be written to the file.

The program in Figure 14-8 illustrates the use of the OS/2 2.1 message APIs:

```
/*                                                                */
/* prog14f8.  OS/2 message APIs.                                  */
/*                                                                */

#define INCL_DOSMISC
#define INCL_DOSNLS
#define INCL_DOSDATETIME
#include <os2.h>

#include <stdlib.h>

VOID DispMessage(PDATETIME pdt, ULONG msgno, PSZ string)
{
   CHAR buffer1[3], buffer2[3], buffer3[4], outbuff[100];
   ULONG ulMsglen;
   PSZ apszMsginsrts[3];

   memset(outbuff, '\0', sizeof(outbuff));

   switch (msgno) {
   case 100:
            apszMsginsrts[0] = string;
            DosGetMessage(apszMsginsrts, 1, outbuff, 100,
                          msgno, "SHOWDATE.MSG", &ulMsglen);
            break;
   case 101:
            apszMsginsrts[0] = _itoa(pdt->day, buffer1, 10);
            apszMsginsrts[1] = _itoa(pdt->month, buffer2,
                                     10);
            apszMsginsrts[2] = _itoa(pdt->year-1900, buffer3,
                                     10);
            DosGetMessage(apszMsginsrts, 3, outbuff, 100,
                          msgno, "SHOWDATE.MSG", &ulMsglen);
            break;
   case 102:
   case 103:
            DosGetMessage(NULL, 0, outbuff, 100,
                          msgno, "SHOWDATE.MSG", &ulMsglen);
            break;
   }

   DosPutMessage((HFILE)1, strlen(outbuff), outbuff);
}

BOOL IsCodePage(CHAR * buffer, ULONG curCP)
{
   PUSHORT pbuf;
   USHORT msgCPcnt, i;
   BOOL cpOK = FALSE;

   pbuf = (PUSHORT)buffer;
   msgCPcnt = *pbuf;
   if (msgCPcnt == 0) return(TRUE);
   pbuf++;
   for (i = 0; i < msgCPcnt; ++i) {
      if (*pbuf == curCP) {
         cpOK = TRUE;
         break;
      }
      pbuf++;
   }
   return(cpOK);
}

INT main( INT argc, CHAR *argv[], CHAR *envp[] )
{
   DATETIME dt;
   CHAR buffer[200];
   BOOL cpOK = FALSE;
   ULONG cplist[3], cpsize, altcp, ulMsglen, i;
   PSZ apszMsginsrts[2];
   CHAR buffer1[4], buffer2[4], outbuff[100];

   /* Make sure the program can display the messages              */
   /* given the code pages installed in the system.               */

   DosQueryCp(sizeof(cplist), cplist, &cpsize);
   DosQueryMessageCP(buffer, 200, "SHOWDATE.MSG", &ulMsglen);
   if (!IsCodePage(buffer, cplist[0])) {
      altcp = ((cplist[0] == cplist[1]) ? cplist[2] :
                                          cplist[1]);
      if (IsCodePage(buffer, altcp)) {

         /* Change the process to use this code page            */
         /* such that the messages are displayed as             */
         /* expected.                                           */
         DosSetProcessCp(altcp);
      } else {

         /* Message file is not valid for the                   */
         /* available system code pages.                        */
         apszMsginsrts[0] = _itoa(cplist[1], buffer1, 10);
         apszMsginsrts[1] = _itoa(cplist[2], buffer2, 10);
         DosInsertMessage(apszMsginsrts, 2,
                          "CODEPAGE=%1,%2", 14,
                          outbuff, 100, &ulMsglen);
         DosPutMessage((HFILE)1, strlen(outbuff), outbuff);
         return(1L);
      }
   }

   for (i = 1;  i < argc;  i++) {
      if (argv[i][0] == '?') {
         /* Display program information                         */
         DispMessage(NULL, 103, NULL);
      } else {
         /* Display unknown parameter                           */
         DispMessage(NULL, 100, argv[i]);
      }
   }

   DosGetDateTime(&dt);

   /* Display today's date                                      */
   DispMessage(&dt, 101, NULL);

   /* Check for Friday the 13th                                 */
   if ((dt.weekday == 5) && (dt.day == 13)) {
      DispMessage(&dt, 102, NULL);
   }

   return(0L);
}
```

Figure 14-8. *Program illustrating OS/2 message APIs.*

The program in Figure 14-8 uses the message definition file shown in Figure 14-6 that is built into a message file named SHOWDATE.MSG.

```
; This is the message input file for component DTE
; These messages will be stored in file SHOWDATE.MSG
DTE
DTE0100E: Unknown parameter %1.
DTE0101I: Today's date is %1/%2/%3.
DTE0102W: Watch your step its Friday the 13th.
DTE0103H: This program displays info about today's date.
```

Figure 15-9. *Message input file.*

The program's output depends on the particular day of the year and configuration of the machine. The program first checks to see if the code page that is in use is one that the message file was built under. If it is not, a check is made to see if the alternate code page can be used. The code pages that are available within the system are set in the CONFIG.SYS file using the SET CODEPAGE=x,y syntax, where *x* is the primary code page and *y* is the alternate code page. If the alternate code page does not match the code pages that are in use, then it may not be possible to display the messages correctly to the user. Therefore, the code pages in the system are displayed and the program terminates. If the messages have been built with an available code page, the program displays a few messages related to the program and the current date.

SUMMARY

Errors of some form will occur when the user is running your program. Handling these conditions gracefully within your program will make it more usable. OS/2 offers several services to assist with error condition management.

Two OS/2 error APIs (DosErrClass and DosError) assist with retrieving more information about an error and disabling OS/2 error pop ups that are not needed by the program.

❑ DosErrClass—This API is used to retrieve other information about an error return code.

❑ DosError—This API is used to disable and enable OS/2 popups that occur during during hard errors and exception errors so the process can handle the error in its own way.

OS/2 defines a class of errors, called exceptions, that can be managed through registering an exception handler. The exception handler is registered for a thread, and when an exception on that thread occurs, the handler is called so that it can manage the exception. The following are the APIs that are used to manage exceptions:

- ❑ DosSetExceptionHandler—This API is used to register an exception handler for the thread.
- ❑ DosUnsetExceptionHandler—This API is used to remove an exception handler that was previously registered for the thread.
- ❑ DosRaiseException—This API is used to cause an exception to occur on a thread.
- ❑ DosSendSignalException—This API is used to cause a signal exception to occur on another process.
- ❑ DosUnwindException—This API is used to remove one or more registered exception handlers.
- ❑ DosSetSignalExceptionFocus—This API is used to indicate which process should be allowed to manage signal exceptions.
- ❑ DosEnterMustComplete—This API is used just before executing part of your program that should not be interrupted by an asynchronous exception. If an exception occurs before the DosExitMustComplete API is called, the exception is deferred until it is called.
- ❑ DosExitMustComplete—This API is used after executing the code that should not be interrupted during an asynchronous exception.
- ❑ DosAcknowledgeSignalException—This API is used to acknowledge that a signal exception has been handled.

When an error occurs, sometimes it is necessary to provide a message to the user to indicate that this happened. OS/2 provides a message management service that allows messages to be stored outside the source code files. These messages can be loaded and displayed during program execution using the OS/2 APIs DosGetMessage and

DosPutMessage. The MKMSGF OS/2 utility is used to create the message file that the messages are extracted from. The following message management APIs were introduced during this discussion:

- ❑ DosGetMessage—This API will load a string from a message file. When loading, it provides a mechanism for substituting information obtained at run time for parts of the message.

- ❑ DosPutMessage—This API is used to display a message in a program that does not run under OS/2 Presentation Manager.

- ❑ DosInsertMessage—This API provides a mechanism for creating a message using string substitution.

- ❑ DosQueryMessageCP—The utility for storing messages (MKMSGF) allows you to store the messages under multiple code pages and languages. This API is used to query the code pages and languages that a message file was built with.

CHAPTER 15

Debugging

"Those who are fond of setting things to rights have no great objection to seeing them wrong." —*William Hazlitt*

INTRODUCTION

You're testing your program and you find out it doesn't do what you expect it to. You might think about it a little, scratch your head, and then glance over the code that is responsible for that feature. Chances are, when you are reviewing your code, you are saying to your self, "I wonder what the value of this variable is at this moment" or "Was this *if* condition true?" Basically, you have decided that you need to know information that is available to you only while the program is executing.

Whenever you're learning a new programming language, through a course or text book, one of the first things they do is teach you how to display information. They usually teach you the infamous hello world program. This program only displays the string *Hello world*. In the C language this is accomplished through the printf statement.

Once you have mastered displaying *Hello world*, you're pretty much in a position to display almost any data. Therefore, if you find yourself wondering about the value of a variable, you can use this function to print its value at a particular moment during program execution.

OS/2 full screen and windowable applications can use this method of program debug. In fact, this method is available to any non-graphical user interface environment. Unfortunately, the printf function can't be used to display information during the execution of an OS/2 Presentation Manager (PM) program.

However, the printf function can be used within PM programs. If stdout is redirected to a file before the PM program is started, the printf information will be written to that file.

Adding printf statements to your program may help you more easily debug your program, but it is less than ideal since this involves having to add temporary code. It's quite cumbersome having to add printf statements since it requires the code to be re-compiled. Once you see the result of the printf, you usually want to add some more to see other results when you have seen that certain conditions are met. There lies the need for a source code debugger.

Source code debuggers are tools, usually provided with a language compiler, that allow the debugging of a running program, using the program's source code. Many source code debuggers also allow debugging with the program's object code as well. Object code debugging is appropriate in solving some problems such as machine hangs.

This chapter introduces the popular debug aids for OS/2 developers. These tools have a great deal of functionality and only a focused discussion of them is provided. The best way to learn about these tools is to use them or watch someone use them. This introduction is provided to make you aware of what is available to help you debug.

SOURCE CODE DEBUGGING

The tool of choice for most debugging is the *source code debugger*. This tool allows you to monitor your program's execution through the program source code. The source code debugger essentially replaces the need for using printf statements to debug with.

The source code debugger allows you to view the contents of variables and data at certain times during program execution. These debuggers offer a number of other capabilities as well.

For the OS/2 32-bit C and C++ languages, IBM offers the IBM C Set ++ Version 2.0 product. This package contains not only the language compilers but other tools to assist with program development. Among this tool set is the IBM C/C++ Debugger.

This debugger is invoked through the IPMD.EXE module. When you invoke this debugger without parameters, it will request the name of the program that is to be debugged followed by any parameters the program needs. You can instead pass this information as parameters with the IPMD command as shown in Figure 15-1.

```
IPMD myprog parameter1 parameter2
```

Figure 15-1. *IPMD sample invocation.*

When IPMD knows what program is to be debugged, it shows its Debug Session Control window and a source code window containing the program's source file with the main() function in it. It scrolls through the file and positions the main() function at the top of the window and highlights the source code line which will first be executed. This highlighted line will show the statement that will be executed next.

The source code window shows executable lines in the color blue and everything else in black. On the left hand side is a prefix area where the source module line numbers are shown. Through the File submenu there is a View option which allows the client window to display source, disassembly, or mixed code. Through the disassembly or mixed views, you are able to step through the object code that was created from the source code.

Program Execution Options

There are a number of options available under the Run submenu of the source code window to control program execution. You can single step one line at a time, step over a function, step into a function, step to the next line that has debug information, or step until a function return occurs.

You can also tell the program to run, run to a location, or jump to a location. Locations are specified by single clicking in the prefix area on the blue line which represents the location.

Running to a location will execute all lines and then stop on the location. Jumping to a location will move the instruction pointer to that location without executing any statements.

There is also an animate option which allows you to have the debugger continuously issue the command to step over a function. This shows you the program's flow in the debugger until you request it to halt.

Breakpoints

The source code window Breakpoints submenu allows you to set several types of breakpoints. A breakpoint causes the debugger process to be activated when the program is running and it encounters the location where the breakpoint was set. The simplest is the line breakpoint, which can be set by double clicking in the prefix area on the line where the breakpoint should be put. The prefix area will change to the color red to indicate that the breakpoint has been set.

Through the Breakpoints submenu, you can set breakpoints on lines, when functions are first entered, with a code address, when a memory location's value changes, or when a DLL is first loaded. Line breakpoints are the most common, but occasionally the others are quite useful.

The debugger can debug code contained in DLLs but the DLL has to be loaded first. If your program issues a DosLoadModule to load a DLL, instead of statically linking to it, you will have to wait to set a breakpoint until the DLL is loaded. By using the load breakpoint, you can break in when the DLL is first loaded so you can set more breakpoints.

Through the Breakpoints dialog, you can further customize breakpoints by specifying which threads the breakpoints are valid for and a frequency rate indicating when to break in. Line, function, and address breakpoints also support testing a program expression to determine if the break should happen. When associating an expression with a breakpoint, such as *id == 500*, your expression can contain references to both local and global variables relative to the location of the breakpoint. The expression is evaluated when the break line is about to be executed and program execution stops only if the expression tests true.

A breakpoint can be disabled, thereby preserving its definition until the breakpoint is needed again.

Program Variables

To view the contents of a variable, you can simply point at the variable name in the source code and double click. This will show a Program Monitor window, which will contain the variable that you requested and its value. Through the Variable submenu, you can choose the Monitor expression choice to type an expression that you want the

Program Monitor window to display. The Program Monitor window will allow you to change the values of a program variable.

Through the Windows submenu, you can select the Local variables option that will display all the local variables for the function the thread is currently running in. There is also a storage option that allows you to show a buffer's content in hex and character format.

The program call's stack and process registers can also be displayed through options in the Windows submenu.

Preparing the Executable

To use the debugger, you don't have to change your source code, but you do have to build your executable differently. When compiling, you must specify the -Ti+ compiler option. It is recommended that optimization (-O-) and inline code (-Oi-) be turned off. You must also specify the /DEBUG option during linking with LINK386. When you specify these options, the executable files will contain the information that the debugger requires about names and statement positions.

OBJECT CODE DEBUGGING

The OS/2 kernel debugger is an object code debugger (assembly language) that can debug both real and protected mode programs. This debugger is convenient for debugging problems in system or application level programs. System level programs include Installable File Systems, device drivers, and the OS/2 kernel. Application level programs include printer drivers, screen drivers, executables, and DLLs.

The kernel debugger is geared toward diagnosing hangs, traps, and memory allocation problems in both system and application level programs. The OS/2 kernel debugger is shipped with the OS/2 2.1 Programmer's Toolkit to assist you in debugging complex problems until source code debuggers can provide adequate debugging functions in complex situations.

The kernel debugger is a version of the OS/2 kernel that contains additional code with a user interface to accept debugger commands. There are two versions of the debug kernel, the HStrict and AllStrict kernels. The Hstrict kernel is simply the original OS/2 kernel with the debugger user interface added. The Hstrict kernel makes it possible to use symbolic addresses, set breakpoints, and trace programs. The AllStrict kernel contains additional error checking code and messages to catch errors and cause an Internal Processing Error (IPE), which the HStrict kernel would let pass or would return as an error to the program.

Along with the AllStrict kernel file, there are Presentation Manager debugging modules that replace the original OS/2 modules. These contain debugging information and messages. These debugging modules help to isolate problems within an application by flagging them as they occur. The debugging messages include warning messages for invalid ranges of parameters, information about resource cleanup as a process terminates, and fatal exits flagging bad handles followed by stack traces. These debugging modules provide a convenient method of checking function errors for all Win and Gpi OS/2 2.1 API calls. The AllStrict kernel results in a very strict operating system from an application's point of view.

In order to use the kernel debugger, you need two machines. One machine has OS/2 with the debug kernel installed. The second machine runs some type of asynchronous emulation package that acts as the debug terminal. The two machines are connected from serial port to serial port (default port is COM2). A local debug setup is connected by means of a null modem cable. A null modem cable is an RS-232 cable that reverses the pin's Transmit Data and Receive Data. It also ties the Request To Send and Clear to Send pins together and ties the Carrier Detect, Data Terminal Ready, and Data Set Ready pins together on a 25-pin to 25-pin cable.

Remote Debug Setup

A remote debug setup is connected by means of a modem. A standard RS-232C cable (no pins reversed) is needed to connect the modem. The debug terminal must dial and connect to the remote modem before beginning the debug session. Therefore, the remote machine with the kernel debugger installed must have the modem set to auto-answer mode. In both local and remote debugging, the asynchronous communications baud rate must be the same as that of the debugger in order to communicate.

The OS/2 2.1 Programmer's Toolkit contains the installation diskettes for the kernel debug. When installing the debugger, be sure to read the READ.ME file located on Debug Support diskette 1 for installation details. The debug installation provides options to install the OS/2 symbol files and utilities, the HStrict kernel, the AllStrict kernel, and the removal of each kernel. Installing the kernal debug files will not affect the execution of your OS/2 programs. The two utilities provided with the kernel debugger are the symbol file creation utility (MAPSYM) and a terminal emulator (T).

Symbolic Debugging

The kernel debugger offers support for symbolic debugging. When a symbol file is loaded into the debugger, you can substitute public symbols used by system or application level programs for memory addresses. This requires that the application provide symbol files for the system or application level modules you are debugging.

The debugger uses the MAPSYM format for the symbol file. The MAPSYM utility executable creates a .SYM file from a .MAP file. The .MAP files are created while LINK386 is running with the /MAP option. To load the symbol files into the debugger, make sure the symbol file name matches that of the module it corresponds to and is located in the same directory.

The debug terminal can use any ASCII terminal emulator software for debugging as long as it can communicate at 9600 bps. The OS/2 2.1 Toolkit's terminal emulator (T) is adequate for debugging.

The debug kernel looks for an initialization file (KBD.INI) which contains commands to be run after the symbol files are loaded. The KBD.INI file is located in the root directory of the boot drive and is a text file that contains any debugger commands each on a separate line. When creating this file, make sure the editor you use appends a CTRL-Z (end-of-file marker) to your file.

Also, make sure the last debugger command is **G** (for Go) or the system will be halted. The OS/2 system editor does not append the end-of-file marker. If you do not use the KBD.INI but need to enter debug commands, enter CTRL-C (enter the debugger) on the debugger terminal and then enter each command. When you have finished, enter the command **G**. Otherwise, the system remains halted.

The following example takes you through debugging a 32-bit application program that contains a Trap E (page fault). Figure 15-2 is the source code of the application (TRAP.C) that contains a page fault. The source code line that assigns *ptr to 5 will trap because ptr is NULL. Although this particular example program could be debugged with a source code debugger, occasionally some traps will occur only when your program is compiled with the optimized (-O+) option and without the debug (-Ti-) option. This is possible because optimized object code is significantly different from object code compiled unoptimized and containing debug data.

```
/*                                                                    */
/* prog15f2.  trap.c                                                  */
/*                                                                    */

#include <os2.h>
#include <stdio.h>

VOID GoTrap()
{
   PUSHORT ptr = NULL;

   *ptr = 5;
   printf("pointer assigned");
}

INT main(VOID)
{

   GoTrap();
   return(0L);
}
```

Figure 15-2. *Source code listing for TRAP.C application program.*

After the kernel debugger and emulator are set up on each system, the first step is to create the file KDB.INI file. Figure 15-3 shows the KDB.INI file.

```
#VSF E
#G
```

Figure 15-3*. Initialization file for kernel debugger KDB.INI.*

The first command, VSF E, sets the trap E vector. This means that if a trap E occurs, the debugger will halt and display the registers. Otherwise, the registers are returned to the screen. The second command, **G**, starts the debugger. For the kernel debugger to use this file, you must reboot your OS/2 machine.

These commands can be entered from the debug terminal by pressing CTRL+C to break into the debugger and then entering the two commands listed in the KDB.INI file.

The next step is to create the symbol file. Using the MAPSYM utility, create the file TRAP.SYM from the map file TRAP.MAP. Figure 15-4 shows the syntax for the MAPSYM utility.

```
MAPSYM TRAP.MAP
```

Figure 15-4. *Syntax for creating a symbol file from a map file.*

Once the symbol file is created, make sure it is in the same directory as the executable TRAP.EXE. Now you are ready to run TRAP.EXE and enter the kernel debugger. Figure 15-5 shows what you would see on the debug terminal after running TRAP.EXE.

```
symbol(s) linked TRAP.SYM
r
Trap 14 (0EH) - Page Fault 0006, Not Present, Write Access, User Mode
eax=00000000 ebx=00000000 ecx=00060010 edx=00060110 esi=00000000 edi=00000000
eip=00010010 esp=00022bdc ebp=00022be4 iopl=2 rf -- -- nv up ei pl nz ac po nc
cs=005b ss=0053 ds=0053 es=0053 fs=150b gs=0000  cr2=00000000  cr3=001c5000
005b:00010010 66c7000500     mov     word ptr [eax],0005    ds:00000000=invalid
```

Figure 15-5. *Outcome of TRAP.EXE execution.*

Notice in Figure 15-5 that the file TRAP.SYM was linked before the CPU registers were displayed. The command **R** will also display these CPU registers from the debug prompt (##). Now that you are in the debugger, run some commands to locate the problem.

First of all, make sure that the trap occurred in the TRAP.EXE executable. The command **.M** will display the module name for a selector. When this command runs, it shows that E:TRAP.EXE is the failing module. This is shown in Figure 15-6.

```
##.M

*har     par       cpg        va     flg next prev link hash hob   hal
0268 %fed0c4fa 00000010 %00010000 1d9 0267 025a 0000 0000 02cd 0000 hptda=02cc
hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
02cd  0268 0000  0838 02b9 02b9  0000 00  00 00 00 shared    e:trap.exe
```

Figure 15-6. *Finding the failing module using the .M command.*

The **.P** command will display information about processes and threads running before the kernel debugger was entered. For additional information on user thread states, use the **.PU** command. This is useful for seeing where the thread left off. Figure 15-7 and Figure 15-8 show the output for the **.P** and **.PU** commands.

```
##.p
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
 001b  0002 0001 0002 0012 blk 0200 7cfcd000 7d1b4020 7d19b9ec      01 pmshell
 000a  0003 0002 0003 0001 blk 0800 7cfab000 7d1b474c 7d199d80      00 harderr
 000f  0003 0002 0003 0002 blk 0800 7cfb5000 7d1b474c 7d19a5dc      00 harderr
 0010  0003 0002 0003 0003 blk 0800 7cfb7000 7d1b474c 7d19a788      00 harderr
 001c  0004 0002 0004 0001 blk 0500 7cfcf000 7d1b4e78 7d19bb98 1eac 10 pmshell
 001d  0004 0002 0004 0002 blk 0200 7cfd1000 7d1b4e78 7d19bd44      10 pmshell
 001e  0004 0002 0004 0003 blk 0200 7cfd3000 7d1b4e78 7d19bef0 1eac 10 pmshell
 001f  0004 0002 0004 0004 blk 0200 7cfd5000 7d1b4e78 7d19c09c 1ecc 10 pmshell
 0020  0004 0002 0004 0005 blk 0200 7cfd7000 7d1b4e78 7d19c248      10 pmshell
 0021  0004 0002 0004 0006 blk 0200 7cfd9000 7d1b4e78 7d19c3f4 1eac 10 pmshell
 0022  0004 0002 0004 0007 blk 0200 7cfdb000 7d1b4e78 7d19c5a0      10 pmshell
 0023  0004 0002 0004 0008 blk 0200 7cfdd000 7d1b4e78 7d19c74c      10 pmshell
 0024  0004 0002 0004 0009 blk 0200 7cfdf000 7d1b4e78 7d19c8f8 1eac 10 pmshell
 0026  0004 0002 0004 000b blk 021f 7cfe3000 7d1b4e78 7d19cc50 1ea0 10 pmshell
 0027  0004 0002 0004 000c blk 0200 7cfe5000 7d1b4e78 7d19cdfc 1eac 10 pmshell
 0025  0005 0002 0005 0001 blk 0400 7cfe1000 7d1b55a4 7d19caa4 1ec8 04 cmd
*002a# 0011 0005 0011 0001 run 073f 7cfeb000 7d1b63fc 7d19d300 19a0 04 trap
```

Figure 15-7. *Current processes running; abbreviated list of the total number of system slots.*

```
##.PU
 Slot  Pid  Ord  pPTDA    Name     pstkframe CS:EIP        SS:ESP        cbargs
 001b  0002 0012 7d1b4020 pmshell  7cfcef4c
 000a  0003 0001 7d1b474c harderr  7cfacf48
 000f  0003 0002 7d1b474c harderr  7cfb6f48
 0010  0003 0003 7d1b474c harderr  7cfb8f48
 001c  0004 0001 7d1b4e78 pmshell  7cfd0f4c d02f:00002581 0017:0000fba6 0008
 001d  0004 0002 7d1b4e78 pmshell  7cfd2f48
 001e  0004 0003 7d1b4e78 pmshell  7cfd4f4c d02f:00002581 0a5f:00007ebe 0008
 001f  0004 0004 7d1b4e78 pmshell  7cfd6f48 005b:1aacb36e 0053:015d3fa4 000c
 0020  0004 0005 7d1b4e78 pmshell  7cfd8f48
 0021  0004 0006 7d1b4e78 pmshell  7cfdaf4c d02f:00002581 0c6f:000027ba 0008
 0022  0004 0007 7d1b4e78 pmshell  7cfdcf48
 0023  0004 0008 7d1b4e78 pmshell  7cfdef48
 0024  0004 0009 7d1b4e78 pmshell  7cfe0f4c d02f:00002581 0ca7:00003e70 0008
 0026  0004 000b 7d1b4e78 pmshell  7cfe4f44 d02f:00000635 0ccf:0199fe1c 000e
 0027  0004 000c 7d1b4e78 pmshell  7cfe6f4c d02f:00002581 0cdf:00003d7a 0008
 0025  0005 0001 7d1b55a4 cmd      7cfe2f44 000f:000044d3 001f:000075b0 000e
*002a# 0011 0001 7d1b63fc trap
```

Figure 15-8. *Additional information on user threads; abbreviated list of the total number of system slots.*

Notice with the **.P** and **.PU** commands that the slot number for TRAP is 002a. Each slot corresponds to a thread. Since each thread has a unique slot number, the kernel debugger can work with any active thread on the system. To change slots, use the **.SS** command. This command is important when you set breakpoints in DLLs. Since a DLL doesn't have a process associated with it, you need to be in the context of the application that is linked with the DLL. For our program, the command **.SS** will verify the slot number that the debugger is using. See Figure 15-9.

```
##.SS
Current task number: 002a
```

Figure 15-9. *Find the current slot number.*

The commands thus far showed which module caused the page fault as well as information about processes and threads running on the system. These are common commands, but they don't focus on the TRAP.C problem. The **LN** command lists the nearest symbol both forward from and backward to the address passed. This command can be used to determine the module and subroutine name where the trap is occurring. Figure 15-10 shows the output from the **LN** command.

```
##LN
005b:00010000 trap:CODE32:GoTrap + 10
005b:00010024 main - 14
```

Figure 15-10. *Find the name of the failing subroutine.*

To verify this, look at the EIP value (00010010) from the CPU registers and look in the map file for this entry point. You will first need to subtract 0x10000, which was specified as the base starting point using the /BASE option when linking the program with LINK386. In the map file (Figure 15-11), 00000010 falls in between 0001:00000000 and 0000000024, which is between GoTrap and main. As the **LN** command has shown above, 10 bytes into the GoTrap routine is where the trap occurs.

```
0001:00000000       GoTrap
0001:00000024       main
0001:00000044       _AllocSpace
0001:000000D8       _printf_ansi
0001:00000100       _printfieee
0001:00000154       _bit_test_set
0001:00000168       _bufprint
0001:0000025C       __dofmto
```

Figure 15-11. *TRAP.MAP file; abbreviated listing of the map file.*

From this information, you could go back to the TRAP.C code and find that the routine GoTrap has code in it that causes this trap. When you compile your program with the -Lf+ option, a .LST file is created that contains the source code and the object code that was produced from it. In this file, you can reference the instruction at address 00000010 to find the instruction that caused the trap.

Other useful commands include stack traces **.K**, dump commands **D**, unassemble **U**, and breakpoints **BP**. The **.K** command traces through the user stack. This is useful for identifying call sequences that led to the trap. Figure 15-11 displays the information for the **.K** command.

```
##.k
005b:0001002f 00022c08 0001167e 00000001 00060110 main + b
005b:0001167e 00000001 00060110 00060010 ffffffff __RunExitList + 96
005b:1a03029b 000002b9 00000000 00030000 00030390 DOS32R3EXITADDR
```

Figure 15-12. *User stack trace.*

The command **D** dumps memory for a specified address range. The DW command dumps the memory in words. Figure 15-13 and Figure 15-14 display the dump information.

```
##DW SS:ESP
0053:00022bdc  0000 0000 0000 0000 2bec 0002 002f 0001
0053:00022bec  2c08 0002 167e 0001 0001 0000 0110 0006
0053:00022bfc  0010 0006 ffff ffff 3744 0001 0000 0000
0053:00022c0c  029b 1a03 02b9 0000 0000 0000 0000 0003
0053:00022c1c  0390 0003 0000 0000 0000 0000 0000 0000
0053:00022c2c  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022c3c  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022c4c  0000 0000 0000 0000 0000 0000 0000 0000
```

Figure 15-13. *Dump memory in words.*

```
##D
0053:00022c5c  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022c6c  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022c7c  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022c8c  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022c9c  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022cac  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022cbc  0000 0000 0000 0000 0000 0000 0000 0000
0053:00022ccc  0000 0000 0000 0000 0000 0000 0000 0000
```

Figure 15-14. *Continue dump from last location displayed.*

The **U** command will unassemble code at a specified address and put it in a mnemonic format. This command allows you to unassemble a number of instructions prior to the address given. Figure 15-15 shows the code at location CS:EIP, and Figure 15-16 shows the code prior to the address.

```
##U CS:EIP
005b:00010010 66c7000500     mov     word ptr [eax],0005
005b:00010015 b800000200     mov     eax,00020000
005b:0001001a e8e1000000     call    _printfieee (00010100)
005b:0001001f 83c404         add     esp,+04
005b:00010022 c9             leave
005b:00010023 c3             retd
trap:CODE32:main:
005b:00010024 55             push    ebp
005b:00010025 8bec           mov     ebp,esp
005b:00010027 83ec00         sub     esp,+00
005b:0001002a e8d1ffffff     call    GoTrap (00010000)
005b:0001002f b800000000     mov     eax,00000000
005b:00010034 eb0a           jmp     00010040
```

Figure 15-15. *Unassemble code at address CS:EIP.*

```
##u cs:eip-10
trap:CODE32:GoTrap:
005b:00010000 55             push    ebp
005b:00010001 8bec           mov     ebp,esp
005b:00010003 83ec08         sub     esp,+08
005b:00010006 c745fc00000000 mov     dword ptr [ebp-04],00000000
005b:0001000d 8b45fc         mov     eax,dword ptr [ebp-04]
005b:00010010 66c7000500     mov     word ptr [eax],0005
005b:00010015 b800000200     mov     eax,00020000
005b:0001001a e8e1000000     call    _printfieee (00010100)
005b:0001001f 83c404         add     esp,+04
005b:00010022 c9             leave
005b:00010023 c3             retd
trap:CODE32:main:
005b:00010024 55             push    ebp
```

Figure 15-16. *Unassemble code prior to address CS:EIP.*

The **BP** command allows you to set BreakPoints. There are two kinds of breakpoints, temporary and sticky. Temporary breakpoints are set when the **GO** command is executed. Once the debugger is entered again for any reason, the temporary breakpoint goes away. Sticky breakpoints, after they are created, have to be removed or disabled by using other debug commands.

Figure 15-17 shows a list of all the kernel debugger commands and a brief explanation of each.

The debug external (dot) commands include the following:

.?	Help for external commands
.B	COM port baud rate
.C	Dump ABIOS common data area
.D	Dump DOS data structures
.I	Swap in page
.it	Swap in TSD
.k	User stack trace
.lm	Print MTE segment table
.ma	Memory arena record dump
.mc	Memory context record dump
.ml	Memory alias record dump
.mo	Memory object record dump
.mp	Memory page frame dump
.mv	Memory virtual page structure dump
.p	Print process information
.r	User register
.reboot	Restart the system
.s	Task context change
.t	RAS trace buffer print

The following is a list of kernel debug commands:

?	Help
bc	Clear breakpoints
bd	Disable breakpoints
be	Enable breakpoints
bl	Breakpoint list
bp	Set a breakpoint
br	Set a debug register
bs	Show time stamp entries
bt	Set a time stamp breakpoint
c	Compare bytes
d	Dump memory
da	Dump ASCII string
db	Dump memory in bytes
dd	Dump memory in dwords
db	Dump GDT entries
di	Dump IDT entries
dl	Dump LDT entries
dp	Dump page directory and page tables
dt	Dump TSS
dw	Dump memory in words
dx	Dump 286 loadall buffer
e	Enter memory
f	Fill memory
go	Go
h	Hexadd
i	Input from port
j	Execute <cmds> if <expr> is true (non-zero)
k	Stack trace
la	List absolute symbols

lg	List active groups
lm	List maps
ln	List near symbols
ls	List all symbols
m	Move
o	Output to port
p	Ptrace
r	Register
s	Search
t	Trace point
u	Unassemble
vc	Clear interrupt and trap vector
vl	List interrupt and trap vector
vt	Add interrupt and trap vector, all rings
vs	Add interrupt and trap vector, ring 3 only
wa	Add active map
wr	Remove active map
y	Display/modify debugger options
z	Default command lines
zl	List the default command
zs	Set the default command

Figure 15-17. *Kernel debug commands.*

The final example takes you through debugging a 32-bit application that loops. Figure 15-18 is the source code of the application LOOP.C. In the GoLoop routine, there is a while loop that always tests TRUE. Without a break or goto instruction within this loop, there is no way the loop will end. This program could be debugged with a source code debugger as well.

```
/*                                                                  */
/* prog15f18.   loop.c                                              */
/*                                                                  */

#include <os2.h>
#include <stdio.h>

ULONG loop;

VOID GoLoop()
{
   while (TRUE) {
      loop = loop + 5;
      loop = loop - 2;
   }
   printf("loop = %ld\n", loop);
}

INT main(VOID)
{
   GoLoop();
   return(0L);
}
```

Figure 15-18. *Source code listing for the LOOP.C application program.*

To start the debugging process, create the symbol file from the LOOP.MAP file and run the LOOP executable. Since this application will not stop the debug terminal, you must enter the debug kernel with CTRL+C. Figure 15-19 shows the debugger after the CTRL+C is entered.

```
eax=0339a316 ebx=00000000 ecx=00060010 edx=00060110 esi=00000000 edi=00000000
eip=00010010 esp=00022be4 ebp=00022be4 iopl=2 -- -- -- nv up ei pl nz na po nc
cs=005b ss=0053 ds=0053 es=0053 fs=150b gs=0000  cr2=0006100b  cr3=001c5000
005b:00010010 a3f00b0200     mov     dword ptr [loop (00020bf0)],eax
                                                            ds:00020bf0=0339a311
```

Figure 15-19. *CTRL-C entered on the debug terminal.*

To find out which module is looping, use the **.M** command. Figure 15-20 shows that the module is E:LOOP.EXE.

```
##.M

*har     par      cpg        va     flg next prev link hash hob   hal
0267 %fed0c4e4 00000010 %00010000 1d9 025c 025b 0000 0000 02ce 0000 hptda=02cc
hob   har hobnxt flgs own  hmte  sown,cnt lt st xf
02ce  0267 0000  0838 02cd 02cd  0000 00  00 00 00 shared    e:loop.exe
```

Figure 15-20. *Find failing module using the .M command.*

Figure 15-21 shows the looping subroutine name using the **LN** command.

```
##LN
005b:00010000 loop:CODE32:GoLoop + 10
005b:0001003c main - 2c
```

Figure 15-21. *Find looping subroutine name using the LN command.*

To find out what thread the debugger has entered, you must use the **.SS** command. Figure 15-22 shows the slot number as 002a.

```
##.SS
Current task number: 002a
```

Figure 15-22. *Find the current slot number using the .SS command.*

Figure 15-23 shows the process and thread information for the current slot. The # sign after the **.P** shows only the information for the current slot and not for the whole system. The current slot is the loop program, which is shown in Figure 15-23.

```
##.P#
 Slot  Pid  Ppid Csid Ord  Sta Pri  pTSD     pPTDA    pTCB     Disp SG Name
*002a# 0012 0005 0012 0001 run 0300 7cfeb000 7d1b63fc 7d19d300 1f48 04 loop
```

Figure 15-23. *Process and thread information for current slot.*

Figure 15-24 shows the unassembled code at address CS:EIP and the unassembled code prior to address CS:EIP.

```
##u cs:eip
005b:00010010 a3f00b0200     mov     dword ptr [loop (00020bf0)],eax
005b:00010015 a1f00b0200     mov     eax,dword ptr [loop (00020bf0)]
005b:0001001a 83e802         sub     eax,+02
005b:0001001d a3f00b0200     mov     dword ptr [loop (00020bf0)],eax
005b:00010022 ebe4           jmp     00010008
005b:00010024 ff35f00b0200   push    dword ptr [loop (00020bf0)]
005b:0001002a b800000200     mov     eax,00020000
005b:0001002f 83ec04         sub     esp,+04
005b:00010032 e8e1000000     call    _printfieee (00010118)
005b:00010037 83c408         add     esp,+08
005b:0001003a c9             leave
005b:0001003b c3             retd
##u cs:eip-10
loop:CODE32:GoLoop:
005b:00010000 55             push    ebp
005b:00010001 8bec           mov     ebp,esp
005b:00010003 83ec00         sub     esp,+00
005b:00010006 8bc0           mov     eax,eax
005b:00010008 a1f00b0200     mov     eax,dword ptr [loop (00020bf0)]
005b:0001000d 83c005         add     eax,+05
005b:00010010 a3f00b0200     mov     dword ptr [loop (00020bf0)],eax
005b:00010015 a1f00b0200     mov     eax,dword ptr [loop (00020bf0)]
005b:0001001a 83e802         sub     eax,+02
005b:0001001d a3f00b0200     mov     dword ptr [loop (00020bf0)],eax
005b:00010022 ebe4           jmp     00010008
005b:00010024 ff35f00b0200   push    dword ptr [loop (00020bf0)]
```

Figure 15-24. *Unassembled code at address CS:EIP and prior to address CS:EIP.*

The information given from the kernel debugger for the executable LOOP.EXE shows that the subroutine GoLoop is where the loop is occurring.

IBM C/C++ Execution Trace Analyzer (EXTRA)

In addition to language compilers and a source code debugger, the IBM C Set ++ Version 2.0 product also contains the Execution Trace Analyzer (EXTRA). Use this tool to better understand what occurs while your application is running. EXTRA creates a trace file containing trace events that occur while your application is running.

Each time a function in your executable is called and returns, a trace event is logged to the trace file. Each trace event contains a time stamp indicating when the event occurred. After creating a trace file, you can then view it in multiple formats that provide various types of information about your program's execution.

Preparing Your Executables for EXTRA

Before EXTRA can gather trace events while your program is executing, your source code must be compiled with the -Gh and -Ti+ options. The -Gh option includes profile hooks that will allow EXTRA to monitor your program. The -Ti+ option adds debug information. The -Ti+ option is required to debug source code by the IBM C/C++ Debugger as well.

Finally, the executables must be linked with the /DEBUG option when running LINK386.EXE. Within the source files that are linked with the executable, you must include the file DDE4XTRA.OBJ. This object file contains the code to hook your functions when running EXTRA.

EXTRA also allows you to trace the OS/2 2.1 APIs. If you want to have these APIs traced, you need to include additional EXTRA LIB files. These LIB files should be specified before the OS2386.LIB specification.

For tracing all the APIs beginning with *Dos*, specify _DOSCALL.LIB. For tracing all the APIs beginning with *Win*, specify _PMWIN.LIB. Finally, for tracing all the APIs beginning with *Gpi*, specify _PMGPI.LIB.

Running EXTRA

To start EXTRA, just run the IXTRA.EXE program. This program first prompts you for the name of the program you want to trace and the command-line parameters it requires. Once EXTRA knows your program information, it displays its Trace Generation window.

The Trace Generation window contains a tree view of the executables that can be traced. Each executable branch contains the OBJ files that can be traced. Each OBJ branch contains the functions within it that can be traced.

The Trace Generation window allows you to pick trigger functions that indicate when the tracing should occur. When you mark a function as a trigger function, tracing begins only when that function is called and ends when the function returns. If no functions are set as a trigger function, the entire program is traced. The Trace Generation window offers other capabilities to help customize your tracing session.

Viewing EXTRA Trace Files

When your program exits, or you request EXTRA to stop, a trace file is produced, and EXTRA asks you in what manner you would like to view it. The Trace Analysis Selection window is displayed requesting that you select one of the trace file viewing options consisting of the Statistics, Call Nesting, Time Line, Execution Density, or Call Graph view.

The Statistics view provides an execution time report for each function called during the trace. For each function traced, the diagram displays the average, minimum, and maximum call time. Other trace statistics are displayed, but the primary purpose of this view is to help isolate slower running sections of your program as well as to determine the execution time of key functions.

The Call Nesting view displays a nested function call diagram in a tree view fashion. The diagram is ordered by how the trace was carried out. For each function called, the tree extends to the right. When a function returns, it comes back to the left. This view is useful for better understanding of what is going on while your program is executing.

The Time Line view shows a nested function call and return sequence. The function time stamp is displayed as well. The time stamp is used for the placement of events on a time dimension axis. This graph will show the chronological relationships.

The Dynamic Call view shows a graphical diagram containing nodes and arcs. Each node is a function, and each arc connecting two nodes represents one function calling the other function. The size of the nodes is larger for functions that have more time spent in them.

When Analyzing the various views that EXTRA displays, you can find areas in your code that could use some tuning to increase performance. You can determine the

sequence of events that leads up to something you didn't expect to occur within your program.
You can also analyze the OS/2 2.1 APIs along with your program. This can provide insight into fine tuning your use of these APIs.

System Performance Monitor/2 (SPM/2)

The IBM System Performance Monitor/2 (SPM/2) Version 2.0 application (sold separately) is designed to give greater efficiency in your OS/2 2.0 environment by helping you manage computer resources. SPM/2 aids in debugging by analyzing the performance of your hardware and software. It collects performance data about system and application resources such as CPU utilization, RAM activity, memory swapping, and detailed information about physical and logical fixed disks.

SPM/2 is a good alternative to object or source code debuggers because the functions that gather and generate reports are better suited to performance issues. SPM/2 provides the following features:

- ❑ Data collection facility
- ❑ SPM/2 graphs
- ❑ A logging facility
- ❑ Theseus2 (memory analyzer)
- ❑ SPM/2 Application Programming Interface (API)

SPM/2 also provides a reporting facility and the ability to perform remote data collection.

Data Collection Facility

The fundamental component of SPM/2 is the data collection facility. This component collects the performance data by interfacing with the OS/2 system. Once the data is collected, the information can be accessed through an API by the logging facility and SPM/2 Monitor, or the performance data can be logged to disk.

The data collection facility can be run on a stand-alone machine or a remote machine by using OS/2 LAN Server and OS/2 LAN Requester.
Remote data collection can help you tune LAN-based applications. The SPM/2 Monitor takes the performance data received from the data collection facility and provides a visual graph for *CPU*, *disk*, and *RAM* usage.

CPU Graph

The CPU graph plots the percentage of time that the CPU is busy against total CPU time. The CPU graph can aid in debugging by identifying the CPU usage of your application. After the CPU graph is running, run your application and check the percentage increase of the CPU usage. If the CPU remains high for an extended period of time, your application may be running too many CPU-intensive tasks.

Disk Graph

The Disk graph displays a data line for each physical disk on your system. Each line is the percentage of time the disk device driver is handling requests. The Disk graph can aid in debugging by measuring your application's disk activity. If the disk activity remains high for an extended period of time, response time for the system will be slow.

RAM Graph

The RAM graph measures the use and allocation of memory and swapping activity. This can be useful when you are debugging to show the amount of memory your application needs. This graph shows fixed memory, working set memory, used memory, and memory swapping in and out. Fixed memory is memory that cannot be swapped out or discarded. As long as the application using the fixed memory is loaded, the memory will remain in physical RAM.

Working set memory includes both fixed and non-fixed memory. Knowing the working set helps to determine if you have enough physical memory for the applications that are currently active on your system. RAM allocated by the OS/2 system without swapping in or out is the used memory.

Swapping out memory is done when the amount of physical memory is exceeded by the demand for physical memory. The memory is swapped to disk. Swapped-in memory is swapped-out memory that is now required in physical memory. If swapping activity remains high, your machine may not have enough physical memory to support the applications currently running.

Logging Facility

The Logging facility creates log files from the data collected from the data collection facility. The report facility then processes the log files and creates detailed reports showing CPU, fixed disk, and memory usage. The CPU activity report consists of process name and ID, percentage and total elapsed time a process was executed, number of dispatches per process, and the time spent executing interrupt handlers while executing.

The physical disk activity provides information for each disk. Physical disk information includes the number of times the disk was accessed as well as the total, average, minimum, and maximum elapsed time spent performing requests. The logical disk activity consists of DosRead/DosWrite requests to the file system including all resources of the file system such as devices, pipes, and files. The RAM activity consists of memory in use, fixed RAM, RAM in the working set, and information about memory swapping. These reports can aid in debugging by giving detailed information about your applications usage of memory and other resources.

Theseus2

Theseus2, or the memory analyzer, reports how much working set memory an application requires. It shows detailed information on OS/2 internals, displays contents of memory, and measures how much memory is actually in use. Theseus2 has three main options to aid in debugging. The options are *system*, *process*, and *registers*.

The system option lists information for the system as a whole. This information includes the working set, RAM usage by process, contents of your SWAPPER.DAT file, and free, idle, and locked memory. Also, there is detailed information concerning the general system and the kernel. The general options include information about device drivers, all modules currently loaded, open files, and more.

The kernel options include kernel memory usage, page frame table, and more. The system options will show detailed information to aid in the debugging of problems with poor performance or memory consumption problems for a system as a whole.

The process option contains information about a selected process. This gives you the opportunity to isolate your application and get detailed information helpful in debugging. Using this option, you can determine the working set required by your application. Also, you can use the LDT option to make sure you are freeing dynamically allocated memory objects. The process option can aid in debugging your application by giving detailed information about a selected process.

The registers options contains information about your processor. It displays iAPX 386/486 control registers, the task state segment, and the current interrupt descriptor table. This option analyzes your hardware in order to give you information about the performance of your hardware.

SPM/2 APIs

SPM/2 provides APIs that allow you to write applications to retrieve performance data and access the Memory Analyzer. Writing applications that use the SPM/2 APIs will be useful for threshold checking and alert generation as well as additional graphing capability for your application. The SPM/2 APIs can also be used to dynamically adjust the way your application runs by using the API to determine the amount of RAM your application has available to it.

SUMMARY

This chapter introduces several debugging tools that are available for use with OS/2 2.1 applications to aid in debugging problems. Both source and object code debuggers are discussed as key tools to aid in typical logic bugs. If you want to write your own debugger, OS/2 2.1 provides the DosDebug API to assist with this. The EXTRA tool provides function trace and timing information to help solve certain types of problems. The SPM/2 tool can help you in tuning a program's working set and in solving memory problems.

CHAPTER 16

Configuration, Installation, and Distribution (CID)

"Marcus, what are you trying to do? Scare me? I don't believe in magic. I'm going after a find with incredible historical significance. You're talking about the bogeyman. Besides, you know what a . . . cautious fellow I am." *Jones, PhD.*

INTRODUCTION

Computers interconnected over Local Area Networks (LANs) are the fastest growing segment of the PC market today. Large companies are *downsizing*, moving their applications and databases from mainframes to LANs. These LANs involve anywhere from a hundred to several hundred workstations. Smaller businesses are discovering the advantages that interconneced PCs have over isolated, standalone workstations.

With the proliferation of LANs, installing and updating applications on computers has become an increasingly important and complex task. Feeding disks into a floppy drive for every new application is no longer a viable option. There must be a way to efficiently and easily automate both the installation and configuration of applications across a LAN. Such a solution must also deal with the heterogeneous environments that exist in the marketplace today.

IBM has advanced a strategy for products, called CID, which stands for Configuration, Installation, and Distribution. Software products that are CID-enabled adhere to guidelines and recommendations set forth for those applications that want to participate in the CID realm. CID-enablement requires a different, though not radical, philosophy for the design and operation of your application's installation and configuration code.

Yet by investing in the time and effort to make your product CID-enabled, you will allow a network administrator to easily install, configure, and maintain your application across all the workstations on the LAN.

Enabling your product for the CID environment does not tie you to a specific distribution mechanism. In fact, CID-enabled applications can participate in a number of different distribution mechanisms, whether by using the LAN Configuration Utility (LCU) component of the IBM Network Transportation Services/2 product (NTS/2) or through the facilities provided by the IBM NetView™ Distribution Manager (NetView DM/2) application. Other vendors are also supporting CID-enabled applications. CID-enabled products can exist in a heterogeneous environment, transparent to whichever Software Distribution Manager (SDM) that your customer uses. (SDM is a generic term for any distribution mechanism that operates in the CID environment.)

Although distribution is an important part of CID, choosing an SDM lies outside of the realm of influence of most software designers. Developers can, however, directly influence the way their product is designed to insure that it can participate in the CID environment. This chapter focuses on what it means for an application to be CID-enabled and how to go about building your application to be an active participant in whatever SDM your customer chooses.

WHAT IS A CID-ENABLED APPLICATION?

Whenever a user installs an application, the user usually performs the task in one of three ways:

❑ *Attended*, which means that the user must sit at the workstation and nurse the floppy drive with disks. (CD-ROMs make this task a little less tedious.) The user might also be required to enter installation or configuration data into panels at different stages of the installation. Whatever the requirement, this installation approach is characterized by manual, time-consuming tasks that the user performs at the target workstation.

❑ *Lightly attended*, which means that the user is not totally divorced from having to physically visit the workstation, but the amount of work the user needs to perform is dramatically reduced. With such an approach, the user can start up the installation program for an application, possibly answer a few questions (such as what drive to install the product on), and walk away. Users can start the installation program by inserting boot diskettes or by executing a program from another workstation that runs on the target workstation.

- *Unattended*, which means the user does not even have to visit the target workstation for the installation of a product. Users could simply schedule the applications for installation on their target workstations through their SDM. This will result in the unattended execution of the installation (or configuration) programs for one or more products on the target workstation. During the installation of a product, the SDM may reboot the workstation as required by each installation program. Successful or unsuccessful installations are reported back to the network administrator as implemented by the SDM.

Note that when the term *installation program* is mentioned, it refers to both the installation and the configuration program of a product. These programs may be separate or they may be combined, allowing one program to perform both the configuration and installation of a product. Also, the terms *target workstation* and *client workstation* can be used interchangeably in this chapter.

The optimal customer solution would be to have all applications installable in an unattended fashion. This would allow the applications on all the workstations in an office to be installed and configured overnight from a remote site and be ready for use in the morning.

However, unattended installation is not always possible, since some workstations may not have the support software to retrieve or communicate information across a LAN. Other factors that exist in heterogeneous LAN and product environments may also prevent unattended installation.

Nevertheless, getting products to a point where users can install and configure them in a lightly attended or unattended fashion will save them a great deal of work, will lower their LAN maintenance costs, and will insure that products and upgrades are quickly and efficiently distributed to the target workstations on a LAN. A product with all these characteristics will be highly prized by network administrators.

CID-enabled products can operate in lightly attended and unattended modes if they support two installation techniques. These techniques are explained more completely later in this chapter, but briefly they are the following:

- ❑ *Redirected installation*—The installation program for the application must be able to retrieve its code images (in such forms as compressed files or disk images) from any drive, whether a network-redirected or a CD-ROM drive. A network-redirected drive physically exists on another workstation on the LAN, but to a client workstation, it looks like a local drive because of the software installed on the client. The key here is to insure that users do not have to use the floppy drives on their workstations to install an application. To a large extent, redirected installation means that a product can be installed from across a LAN.

- ❑ *Response files*—These ASCII files, with their specific *keyword and value* syntax, replace the need for the user to input installation and configuration values manually at install time. Response files remove the need for users to know what the installation and configuration parameters mean. The LAN administrator can provide a specific response file that tailors parameters, such as the maximum database buffer size, for a group of workstations or a single workstation. A CID-enabled product will use a response file for its installation and configuration process. Response files do not eliminate the need for a product's regular installation program. Users should still be able to install and configure their product through a windowed interface, if so desired.

Once a product supports redirected installation, response files, and a few additional command-line parameters for its installation program, the application has suited up as a player in the CID arena. The code example later in Figure 16-3 shows one way to design your installation program to be CID-enabled.

There's a difference between a product's installation program and its regular program (which may consist of several programs—the total of which is called the application). The installation program is used to install or configure the program from images or files that reside on a disk or code server. It is typically run only once or very infrequently to establish or configure a program on a workstation. The user invokes the application program files to use the actual product.

Note also that the configuration of a product need not take place at installation time. For example, users may want to change their configuration values for tuning purposes after installing and testing applications across their LAN network.

THE CODE SERVER AND ITS DIRECTORIES

The term *code server* is used frequently when describing how an installation program can retrieve both the response files and product code files (whether in disk image or compressed format) for installation on a target workstation.

A *disk image* is a binary file containing the contents of an entire disk. *Compressed files* are created by compaction programs. The code server acts as a central repository that the installation program can access using redirected installation.

One such recommended directory structure for products available on a code server is presented in Figure 16-1:

```
\CID
  \CLIENT (stores the client OS/2 command files, such as those
       used by the LAN CID Utility)
  \DLL (stores DLL files, generally those used in the EXE
       directory)
  \EXE (stores executable programs files)
  \IMG (stores the disk images or product files - each
       subdirectory pertains to a specific product)
     \PROD1
     \PROD2 (etc.)
  \CSD (stores the Corrective Service Disk or Service Pak
       images for a product - each subdirectory pertains to a
       specific product)
     \PROD1
     \PROD2 (etc.)
  \LOG (stores the log and history files for a product - each
       client subdirectory pertains to a specific product)
     \PROD1
     \PROD2 (etc.)
  \RSP (stores the response files for a product - each
       client subdirectory pertains to a specific product)
     \PROD1
     \PROD2 (etc.)
```

Figure 16-1. *CID directory structure.*

The response files would have names like NODE0001.RSP and NODE0002.RSP, for example, where the file names are the same as the client node names. The filename \CID\RSP\LS30\NODE0001.RSP would indicate to the user that the response file was for the target workstation named NODE0001 and that the response file was specific to the LAN Server 3.0 product.

The command files in the \CID\CLIENT\ directory are used as part of the SDM target (or client) installation process to identify the products that need to be installed, the order of installation, and the commands needed to complete the installation. These client command files can indicate that Product X be installed before Product Y. The SDM that the user chooses may require its own unique directory structure format. With support for response files and redirected installation, your application need not concern itself with the directory structure or the functions of the SDM, yet it will still be able to participate in the CID environment.

REDIRECTED INSTALLATION

To design your application to support redirected installation, your installation program will need to support two functions:

- ❑ The ability to transfer the images or contents of the program installation disks to a hard drive or to another medium. This allows network administrators to place your product images on their code server.

- ❑ The ability to retrieve the disk images or program files from a network, redirected, or non-floppy drive during the installation process.

Redirected installation does not mean that your program cannot be installed from a floppy drive. In fact, being able to install from the A or B drive is still of paramount importance to the large base of standalone workstations. Supporting the installation of your program from a floppy drive is still a primary requirement for most applications. Yet by extending your installation program to support redirected installation, you will extend the reach of your program to both the standalone and the LAN environments. You will also make your application more attractive to LAN administrators and users.

Note that an application could use an operating system utility program, such as XCOPY, to transfer program files from its product disks to a code server. It can also use XCOPY to transfer files from a network or redirected drive to the target workstation. You don't need to build a special function to accomplish this.

Applications, including those not specifically designed to be CID-enabled, have already started to support redirected installation. Typically, this takes the form of allowing the user to copy the contents of the program disks to a hard drive or a network partition. Afterwards, when the user starts the installation program, it will prompt the user at the beginning to ask for the drive and directory path from which to retrieve the program files. These programs also ask for the target partition and directory in which to install the application.

More sophisticated installation programs will copy the disk images to predetermined directories on a target disk partition. These programs will even create the directory structures if they do not exist. Whether the user performs an XCOPY on the disks or uses the installation program to transfer files, supporting redirected installation helps move users away from having to perform attended installation.

RESPONSE FILES

The main purpose of response files is to provide predetermined answers to all the possible installation and configuration prompts that the installation program may ask the user. This frees the user on the target workstation from having to know what the parameters are, their ranges, and so on.

Someone, perhaps the network administrator or an expert with the application who knows how to tune the parameter values for optimal performance, can build the response file for the client workstation before the installation program is even invoked.

Again, response files do not replace the need for panels that prompt the user for values or ask the user to select from a list of choices. First-time users may prefer this route with drop-down lists, context-sensitive help, and other user-friendly interface controls, instead of leafing through a manual or installation guide. Response files may be useful, but they are not a requirement for standalone users, who are not likely to install the program more than a few times.

There are some programs, very few in number, that do not require the user to specify any installation or configuration parameters. (Even most of today's entertainment software will ask you to choose the sound cards and video displays on your system to confirm or override any default selections it may have detected.) The net result is that without support for response files, your program may not be able to fully support lightly attended installation. For programs that require parameters, your application certainly will not be able to support unattended installation.

Response files are ASCII files that a user can freely edit. Each line in the file has a maximum length of 255 characters. The most basic unit of a response file is the keyword=value pair. For example, the following entries in a response file specify the configuration values for the maximum number of database connections and the database buffer size:

```
DB_MAX_CONNECTIONS = 254
DB_BUFFER_SIZE = 64
```

Note that response files can contain keyword assignments that are specific to both installation and configuration. It is up to the installation program to determine if such a distinction needs to be made. This means that the program could, for example, accept and process all the installation and configuration parameters in a response file when the program is first installed.

If the user or administrator runs the installation program again and if the program detects that the program already exists on the target machine, the installation program could still use the response file. It would, however, ignore those parameters that only pertained to installation. If the user specified a reinstallation of the program, the installation keywords would then be used.

RESPONSE FILE SYNTAX

The following rules and recommendations govern the syntax of response files:

- ❑ You need a keyword for every installation or configuration parameter that the user can specify. Complete coverage is essential.

- ❑ Keyword names should not be case sensitive. For example, SESSION_ID must be equivalent to seSSIon_ID. They can begin with numbers or letters, but not asterisks or semicolons. Keywords cannot have any blanks or keyword characters within them. Use the underscore character (_) to help make the keywords more readable.

- ❑ Keywords must be unique within a product. There should be no ambiguity in cases where a keyword is used for both installation and configuration, yet it has two different meanings depending on whether the program is being installed or configured.

- ☐ If a keyword was not specified in the response file, the installation program should either assign it a default value or log an error and end the program. If the installation program used a default value, it may want to log a warning message indicating this.

- ☐ If duplicate keywords are specified in a response file, take the value of the last one. For example:

  ```
  SCREEN_COLOR = BLUE
  SCREEN_COLOR = RED
  ```

  Here, the installation program would assign red to the screen color after it finished processing the response file. This is the recommended precedence.

- ☐ Everything after the equals sign following the keyword should be assigned to the keyword, and there need not be a blank space between the keyword and the equal sign or between the equals sign and the value. For example, the following will assign = 3 to SESSIONS:

  ```
  SESSIONS==3
  ```

  The user has flexibility in specifying values. Strings do not need to be enclosed in quotes, and numbers do not need to be prefixed with a unique character. The installation program will need to perform the parsing, range checking, and any string or number conversions. The principle is to make life easy for the user.

  Note that keywords need not require an equals sign or value to follow it. In such cases, you need to make clear to the user what will be done when the installation program encounters a lone keyword. Is the keyword instead a command? Does this mean that the default value will be used upon installation?

  If the program has already been installed and configured, will the absence of a value mean that an error occurred, that the keyword will be reset to the default value, or that the function belonging to the keyword is no longer needed? Will a keyword followed by an equals sign, but without a value, indicate a null string? Your installation program will need to make these choices and document its course of action in such cases; keyword and value semantics are defined by your product.

For example, if the TOOLSGUI keyword was included in the response file, it could mean that the user wanted to install the TOOLSGUI subprogram with the application. If the keyword was absent, this might mean that the user did not want to include it upon installation. It might also mean that the user wants to remove it the next time the installation or configuration program is run.

- ❑ A value cannot span multiple lines, except in the case of lists, which act like array data structures. Figure 16-2 shows an example of lists in action:

```
MODULE = (
           NAME = UNIT1
           MAX_LENGTH = 42
          )
MODULE = (
           NAME = UNIT2
           MAX_LENGTH = 67
          )
```

Figure 16-2. *Lists spanning multiple lines.*

These lists indicate ownership to the parental keyword. The first module has a different name and maximum length than the second module. The placing of the parentheses is important. The left parenthesis must be placed after the equals sign of the parent keyword and it must be the last non-white character on the line. (Note the difference between non-white and non-blank. Non-white characters can include tabs.) The closing right parenthesis must also be on a line of its own. This makes the response file a little more readable for the user and makes the job of parsing a little easier for the installation program.

Nested lists are allowed too, but most programs will not need such complex constructs.

Note that lists prevent both you and the user from having to use long, contrived names, like MODULE_1_MAX_LENGTH. These constructs act like list arrays that come in handy when the user needs to specify a variable number of similar items.

In the above example, to remove a module that has already been installed, the user could include the DELETE keyword along with a unique and required keyword and value. The installation will then interpret the DELETE to mean remove the module from the product's configuration. To remove a module named UNIT1 that was previously installed, users could specify the following in their response file:

```
MODULE = (
           NAME = UNIT1
           DELETE
          )
```

❑ Comments begin with an asterisk (*) or semicolon (;). They are the first non-white character in a comment line. For example:

```
* This is a valid comment line.
; This too
But this isn't
BUFFER = 128 * Are you kidding?
```

In the last example line, the user meant for there to be a comment to describe the buffer parameter. Instead, *128 * Are you kidding?* will be assigned to the BUFFER keyword. The installation program will inevitably reject the value and log it as invalid.

❑ Response file names should have .RSP extensions. This applies to both group and client response files, as explained in the next section.

GROUP AND CLIENT RESPONSE FILES

When duplicate keywords are found in a response file, the recommended processing scheme states to take the last value specified. Because of this scheme, a division between group and client response files is natural and is easy to implement. A *group response file*, sometimes called a *supplemental* or *general* response file, may contain a set of keyword and value assignments that could be applied as defaults to all the workstations in a specified group, such as all those on a single LAN, department, or floor.

The keyword and value assignments in a *client response file* are based specifically on the requirements or uniqueness of its target workstation. Keyword values can override those specified in a group response file when the group file is processed first. Network administrators typically create the group response files for all the products that will be installed and configured for a logical group of workstations. They can also create all the unique client response files too.

These client files afford both the user and the network administrator the opportunity to make specific changes to a single workstation. Together, the group and client response files for a product can make that installation and configuration unique for the application on the target workstation.

Group and client response files have no special distinguishing features. They follow the same guidelines and recommendations for response files; they differ only in their intended use. Users can specify the INCLUDE standard keyword to incorporate a group file at the beginning of a client response file. The set of recommended standard keywords for response files is described in the next section.

STANDARD RESPONSE FILE KEYWORDS

Along with the keywords that your program defines for installation and configuration, support for three others is recommended. Called *standard keywords*, they perform actions that are not directly related to an installation or configuration keyword value in your program. You may not need any of the standard keywords or you may want to add to this special set according to the requirements of your installation program.

The basic standard keywords defined are the following:

❑ INCLUDE—Includes the contents of another response file at the point where this keyword is encountered in the current response file. The syntax is as follows:

```
INCLUDE = drive:\pathname\filename or filename
```

Users can use the INCLUDE keyword to bring in the keywords and values of a group response file at the beginning of a client response file. Thus, with the processing order of taking the last value of any duplicated keywords, the values in the client file could override those specified in the group file. Note, however, that INCLUDE calls can be specified anywhere in the response file, whether a group or client file. Beware of a cyclical call where response file A includes B and B includes A. Your installation program should detect and prohibit these cycles.

- ❑ COPY—Copy a source file to a target file. The syntax is as follows:

  ```
  COPY = sourceFilename targetFilename
  ```

  Users should be able to specify this standard keyword at any point in their response file, but it is up to the application developer to decide what constraints, if any, should be harnessed on the placement, syntax, and use of the keyword. For example, should the installation terminate if the target file already exists? Users will find the COPY keyword handy when they want to back up a file, such as the PROTOCOL.INI or CONFIG.SYS file, before a reinstallation or configuration takes place.

- ❑ USEREXIT—Runs a user exit program. The syntax is as follows:

  ```
  USEREXIT = executableFileName
  ```

  User exit programs allow the administrator to create a separate, custom program that performs processing external to the installation program or the SDM process. The executable file name can be either an .EXE or a .CMD file that can be run in the command processor shell (CMD.EXE). Developers are encouraged to tailor the use of this keyword to the needs of their program. A standard method for searching for the correct file for both the INCLUDE and USEREXIT keywords is encouraged. One suggested algorithm follows:

  1. Get the fully qualified filename, if specified. If wildcard characters are allowed, get the first filename that matches.

  2. If the fully qualified filename is not specified, search the current directory for the file.

  3. If not found, search the PATH and then the DPATH OS/2 environment variables.

  4. If the file is still not found, the installation program has the option of logging a warning message and continuing or else throwing up its hands,

logging an error message, and terminating. The latter route has a practical advantage to the user, since the user may be relying on the user exit program to provide a crucial task.

An easy way to implement the user exit program in your installation program would be to call the DosExecPgm API. With this API you can run the CMD.EXE command shell and send it the user exit executable file name as its parameter. However, this may not work if the installation program runs under Presentation Manager and the user exit program does not. In such a case, the CMD.EXE shell would have to be started in another session using the DosStartSession API. Most installation programs will not have to deal with the complexities of another session, since they will run as a command-line program.

INSTALLATION PROGRAM COMMAND-LINE PARAMETERS

The following is a list of optional, but recommended, command-line parameters that your installation program should support for CID enablement. They could be expanded to include those that are specific to a product, but these are the recommended parameters:

- ❑ /S:sourceDirectory—Specifies the full directory to the product's disk images or program installation files on the code server. If this parameter is not specified, the installation program will search for the source files in the default directory on the code server.

- ❑ /R:clientFilename—Specifies the full path name to the client response file on the code server. Again, if this parameter is not specified, the installation program will search for the client response files in the default directory on the code server. (Note that the user could invoke the installation program from a boot disk and indicate that the program should retrieve its response files from the disk in the floppy drive.)

- ❑ /G:groupDirectory—Specifies the full directory to the group response file on the code server. As mentioned earlier, this allows the user to utilize the

INCLUDE keyword to specify a filename that the installation program would look for in the group directory. If the group directory was not specified, the installation program can search for the file indicated by the INCLUDE keyword in the same fashion as the search for the user exit program.

- ❑ /(L1-L5):logFileName—Specifies the full path name to the log file on the code server or the client workstation. The installation program can dynamically update the log file on the client or server workstation or can copy the log file from the code server to the client when the installation is complete. The user could specify up to five different log files (/L1:, /L2:, ... /L5:). However, if they are needed, installation programs should use /L1: and /L2: as error and history files respectively.

  Log entries could be appended to the file, or they could replace entries recorded by a previous installation. If entries are appended to a log file, the installation program may want to prune the oldest entries once the log file reaches a certain size, such as 64K. The user may, however, want to keep a full history of a log. Log files should also have a .LOG extension.

- ❑ /T:targetDirectory—Specifies the full directory in which the product should be installed on the client workstation. If this directory is not specified, the installation program would use the default target directory.

- ❑ /TU:configsysDirectory—Specifies the full directory on the target workstation in which the CONFIG.SYS file resides. If this directory is not specified, the installation program can search for the default CONFIG.SYS file on the boot drive of the workstation.

  The installation program can use this parameter to find the correct CONFIG.SYS file for updating. (Your installation program may need access to the CONFIG.SYS file to include another DEVICE= statements or append a new directory path to the PATH environment variable.) Multiple CONFIG.SYS files may exist on a machine if it has been enabled for dual boot, for example.

The above parameters are merely recommended ones for you to use with your installation program. By implementing them, you provide a common interface that users can recognize to easily install their products.

You may want to support a combination of parameters and keywords. For example, you could support the /T: parameter to specify the client's target directory, but you may also have a TARGET_DIRECTORY keyword and value assignment in your group or client response file that would override your command-line parameter.

In such cases where the user has multiple places where an installation or configuration keyword value can be defined, the following processing order is recommended:

1. Client and group response file. Take the last value for duplicated keywords in the resulting combined file.

2. If values were not specified in the response files, use the existing values that were specified by the user from a previous installation. Use the values that are already set in the product's .INI file, for example.

3. If none of the values are available from the above methods, use the default values supplied by your installation program. (For every optional installation or configuration keyword, have a default value, if possible.)

RETURN CODES

Your installation program needs to supply a return code to the SDM upon termination. This can indicate that the installation was successful or that an error occurred, such as the disk being full. The return code can also indicate whether the SDM should perform an IPL (a.k.a. Ctrl-Alt-Del) after receiving the return code from your installation program and if the SDM should reinvoke your installation program after an IPL. (Some products need to re-IPL their target workstation before their installation can be considered complete. For example, the OS/2 documentation recommends that users reboot their machines after initially installing OS/2.) The recommended return codes in 2-byte hex format are presented in Table 16-1:

RETURN CODE	MEANING
00 00	Successful termination.
08 00	The data object was not found.
08 04	Could not access the data object since it is already in use.

RETURN CODE	MEANING
08 08	Could not access the data object since the proper authorization was not provided.
08 12	Could not find the path to the data object.
08 16	Product was not configured.
12 00	An I/O error occurred.
12 04	The device was not ready.
12 08	Disk full.
16 00	The program was not invoked correctly. (For example, the installation program was not called with the proper parameters, or a parameter was invalid.)
16 04	An unexpected condition was encountered. (The mother of all return codes.)
FE 00	Successful termination. Re-IPL. (No callback.)
FE 04	Successful termination but with warning messages. Users should consult the warning log. Re-IPL. (No callback.)
FE 12	Successful termination but with severe error messages (definitely not good). Users should consult the error log. Re-IPL. (No callback.)
FF xx	Successful execution. Re-IPL and call the installation program again. The xx hex value indicates what the state the installation program should proceed to when it is called again. Your installation program defines xx, since it has to interpret the code when it regains control after the IPL.
FD xx	Reserved by the SDM.

Note how the prefix relates to the meaning :

PREFIX	MEANING
0x00	Success!
0x08	Problems obtaining or configuring data objects.
0x12	I/O or device problems.
0x16	Bad invocation or an unexpected error.
0xFE	The program terminated and the target machine is re-IPLed. The second set of return code hex numbers needs to be examined to see if the installation program was completely successful or if a warning or error occurred.
0xFF	Program executed okay (as opposed to terminating); however, the workstation needs to be re-IPLed, and the installation program needs to be called again with the second set of return code hex numbers.

Table 16-1. *CID return codes.*

How can the installation program regain control after an IPL? It's automatic on a callback return code. The most common way is for the SDM to place a call to the installation program in the STARTUP.CMD file. One of the parameters to the installation program would be the second set of hex numbers. This way when the SDM re-IPLs the workstation, the SDM gets reinvoked when the STARTUP.CMD file is run.

CID EXAMPLE PROGRAM

The following example program illustrates how an installation program can retrieve and interpret CID-related command-line parameters and an installation response file. The sample program PROG16.C in Figure 16-3 retrieves and processes command line parameters, reads the response file NODE0001.RSP, and saves the installation values in the CIDEXMP.INI file. The response file is shown in Figure 16-4. Not every CID command-line parameter is used in this example. The program writes both the history and error information into a log file and returns the CID return code when it exits. If a history file exists, it will append the new messages to the end of the file. Figure 16-5 shows a generated log file.

This program simulates the installation process of a program. It does not include all of the error checks that you will want to put into your program. It also creates a new INI file each time it runs. You will want to use the old values in the INI file, if available, as well as back up the file in case something goes awry when you are creating the file.

The program recognizes two keywords, BUFFER and TIMEOUT. The values retrieved from the response file were not checked for their range or syntax. You will want to do both for each value in your supported response file.

Note that one of the default directories for the code server hangs off the theoretical Y: drive partition. The program will be able to access this drive through redirected installation. (The SDM that the customer uses will establish the connection to the Y: drive.) You can change the program to dynamically use the target drive letter specified by the user or the SDM.

Figure 16-3 provides an example:

```
/**********************************************************************/
/* prog16.c.  CID installation program example.                       */
/**********************************************************************/

#define INCL_DOSMEMMGR          /* Include Memory Management Calls */
#define INCL_DOSFILEMGR         /* File system values              */
#define INCL_DATETIME
#define INCL_NOPMAPI            /* Since PM APIs are not used,     */
                                /* don't include them              */

#include <os2.h>
#include <stdio.h>
#include <bsememf.h>

/* CID return codes                                                   */

#define RC_SUCCESS          0x0000
#define RC_IOERROR          0x1200
#define RC_PARMERROR        0x1600
#define RC_UNEXPECTEDERROR 0x1604

/* Stores command line parameter values                               */
```

```
typedef struct _cmdParms
{
   UCHAR  sourceDir[256];
   UCHAR  clientFile[256];
   UCHAR  groupDir[256];
   UCHAR  log1[256];
   UCHAR  targetDir[256];
   UCHAR  configSysDir[256];
   UCHAR  nodename[9];
} CMDPARMS;

/* Structure for the response file keywords and values              */

typedef struct _instParms
{
   UCHAR  bufferStr[55];
   UCHAR  bufferVal[256];
   UCHAR  timeoutStr[55];
   UCHAR  timeoutVal[255];
} INSTPARMS;

/* Allowable command line parameters                                */

static UCHAR stdCmdLineArg[10][5] = {"/S", "/R", "/G", "/L1",
   "/T", "/TU"};

/* Pre-declaration of functions                                     */

void trimBlanks(CHAR * );
int splitKeywordValName(UCHAR *, UCHAR *, UCHAR *);
int stringcpyFromTo(PSZ, PSZ, int , int);
BOOL isCommentLine(UCHAR *);

/******************************************************************/
/* Main program                                                   */
/******************************************************************/

int main (USHORT argc, PCHAR argv[])
{
   int i, j, k,charPos;
   UCHAR cmdLineArg[10][50], tempStr[300], fileStr[256], dirStr[256];
   UCHAR keywordStr[100], valueStr[100];
   UCHAR lineStr[300];
   CMDPARMS cmdLineParms;
   INSTPARMS instParms;
   PSZ currparm;
   ULONG parmfound;
   ULONG keywordfound;
   ULONG ObjSize=3000;
   DATETIME DateTime;
   APIRET rc;

   /* File variables                                                */
   HFILE FileHandle,log1FileHandle,dupFileHandle;
```

```
ULONG bytesRead, bytesWritten;
ULONG ActionTaken;
UCHAR filename[20];
ULONG filePtr;
UCHAR *rspFileBuffer;

/* Initialize strings and buffers                              */

strcpy(fileStr,"");
strcpy(dirStr,"");
strcpy(keywordStr,"");
strcpy(valueStr,"");
strcpy(instParms.bufferStr,"BUFFER");
strcpy(instParms.bufferVal,"");
strcpy(instParms.timeoutStr,"TIMEOUT");
strcpy(instParms.timeoutVal,"");

/* set up the command line defaults                            */
strcpy(cmdLineParms.sourceDir,"Y:\\CID\\IMG\\PROD\\");
strcpy(cmdLineParms.clientFile,"");
strcpy(cmdLineParms.groupDir,"");
strcpy(cmdLineParms.log1,"");
strcpy(cmdLineParms.targetDir,"");
strcpy(cmdLineParms.configSysDir,"C:\\");
strcpy(cmdLineParms.nodename,"NODE0001");

/* get the command line parameters                             */

for (i=1;i < argc;++i) {
   parmfound = FALSE;

   /* /S:sourceDirectory                                       */
   if (strnicmp(currparm,"/S:",3) == 0) {
     stringcpyFromTo(currparm,&cmdLineParms.sourceDir,3,
        strlen(currparm));
     parmfound = TRUE;
   }
   /* /R:clientFilename                                        */
   if (strnicmp(currparm,"/R:",3) == 0) {
     stringcpyFromTo(currparm,&cmdLineParms.clientFile,3,
        strlen(currparm));
     parmfound = TRUE;
   }
   /* /G:groupDirectory                                        */
   if (strnicmp(currparm,"/G:",3) == 0) {
     stringcpyFromTo(currparm,&cmdLineParms.groupDir,3,
        strlen(currparm));
     parmfound = TRUE;
   }
   /* /L1:historyAndErrorLogFile                               */
   if (strnicmp(currparm,"/L1:",4) == 0) {
     stringcpyFromTo(currparm,&cmdLineParms.log1,4,
        strlen(currparm));
     parmfound = TRUE;
```

```
      }
      /* /T:targetDirectory                                            */
      if (strnicmp(currparm,"/T:",3) == 0) {
        stringcpyFromTo(currparm,&cmdLineParms.targetDir,3,
           strlen(currparm));
        parmfound = TRUE;
     }
      /* /T:configsysDirectory                                         */
      if (strnicmp(currparm,"/TU:",4) == 0) {
        stringcpyFromTo(currparm,&cmdLineParms.configSysDir,4,
           strlen(currparm));
        parmfound = TRUE;
     }
      /* /N:nodename                                                   */
      if (strnicmp(currparm,"/N:",3) == 0) {
        stringcpyFromTo(currparm,&cmdLineParms.nodename,3,
           strlen(currparm));
        parmfound = TRUE;
     }
     if (parmfound == FALSE) {
        printf("Error:  Invalid parameter: %s\n",currparm);
        exit(RC_PARMERROR);
     }

   }

 /* Put default values in that need to be calculated from other      */
 /* values,if they weren't specified on the command line.            */

   if (strcmp(cmdLineParms.clientFile,"") == 0) {
      strcpy(tempStr,"");
      strcat(tempStr,cmdLineParms.nodename);
      strcat(tempStr,".RSP");
      strcpy(cmdLineParms.clientFile,tempStr);
   }

   /* log file                                                         */
   if (strcmp(cmdLineParms.log1,"") == 0) {
      strcpy(tempStr,"");
      strcat(tempStr,cmdLineParms.nodename);
      strcat(tempStr,".LOG");
      strcpy(cmdLineParms.log1,tempStr);
   }

   /* Put header in log file.                                          */

   rc = DosOpen(cmdLineParms.log1,
                &log1FileHandle,
                &ActionTaken,
                0L,                                     /* File Size */
                FILE_NORMAL,
                FILE_OPEN | OPEN_ACTION_CREATE_IF_NEW,
                OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE,
                NULL );                       /* No extended attributes */
```

```
if (rc != 0) {
   printf("Error:  DosOpen for cmdLineParms.log1
      with rc = %ld\n",rc);
   exit(RC_IOERROR);
}

rc = DosSetFilePtr(log1FileHandle, 0, FILE_END,
    &dupFileHandle);

if (rc != 0) {
   printf("Error:  DosSetFilePtr for cmdLineParms.log1
      with rc = %ld\n",rc);
   exit(RC_IOERROR);
}

DosGetDateTime(&DateTime);
sprintf(tempStr,"\n\r%2.2d-%2.2d-%2.2d %2.2d:%2.2d:%2.2d -
    Started installation for node %s.\n\r",
    DateTime.month,DateTime.day,DateTime.year,DateTime.hours,
    DateTime.minutes,DateTime.seconds,cmdLineParms.nodename);
rc = DosWrite(log1FileHandle,tempStr,
   strlen(tempStr),&bytesWritten);
sprintf(tempStr,"   Source Directory - %s\n\r",
   cmdLineParms.sourceDir);
rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
   &bytesWritten);
sprintf(tempStr,"   Client Filename - %s\n\r",
   cmdLineParms.clientFile);
rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
        &bytesWritten);
sprintf(tempStr,"   Group Directory - %s\n\r",
   cmdLineParms.groupDir);
rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
   &bytesWritten);
sprintf(tempStr,"   Error Log Filename (L1) - %s\n\r",
   cmdLineParms.log1);
rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
   &bytesWritten);
sprintf(tempStr,"   History Log Filename (L2) - %s\n\r",
   cmdLineParms.log1);
rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
        &bytesWritten);
sprintf(tempStr,"   Target Directory - %s\n\r",
   cmdLineParms.targetDir);
rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
   &bytesWritten);
sprintf(tempStr,"   CONFIG.SYS Directory - %s\n\r",
   cmdLineParms.configSysDir);
rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
   &bytesWritten);
sprintf(tempStr,"   Nodename - %s\n\r",
   cmdLineParms.nodename);
rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
   &bytesWritten);
```

```
/* read in the response file                                        */

strcpy(filename,cmdLineParms.clientFile);

rc = DosOpen(filename,
             &FileHandle,
             &ActionTaken,
             0L,                                       /* File Size */
             FILE_NORMAL,
             FILE_OPEN,
             OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE,
             NULL );                         /* No extended attributes */

if (rc != 0) {
   sprintf(tempStr,"Error:  Could not open response file %s with
      rc = %ld\n\r", filename,rc);
   rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
      &bytesWritten);
   rc = DosClose(log1FileHandle);
   exit(RC_IOERROR);
}

if ((rspFileBuffer = (UCHAR *) malloc(ObjSize)) == NULL) {
   /* memory allocation error                                       */
   exit(RC_UNEXPECTEDERROR);
}

strcpy(rspFileBuffer,"");
rc = DosRead(FileHandle,rspFileBuffer,ObjSize,&bytesRead);

if (rc != 0) {
   sprintf(tempStr,"Error:  Could not read response file %s with
      rc = %ld\n\r", filename,rc);
   rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
      &bytesWritten);
   rc = DosClose(log1FileHandle);
   free(rspFileBuffer);
   exit(RC_IOERROR);
}

rc = DosClose(FileHandle);

if (rc != 0) {
   sprintf(tempStr,"Error:  Could not close response file %s with
      rc = %ld\n\r", filename,rc);
   rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
      &bytesWritten);
   rc = DosClose(log1FileHandle);
   free(rspFileBuffer);
   exit(RC_IOERROR);
}

/* parse each line in the response file                             */
```

```
charPos = 0;
while (charPos <= bytesRead) {
   i = 0;
   strcpy(tempStr,"");
   while (rspFileBuffer[charPos] != '\n') {
      if (charPos == bytesRead)
         break;
      tempStr[i++] = rspFileBuffer[charPos++];
   }
   charPos++;
   tempStr[i] = '\0';

   /* If the line is not a comment, get its keyword             */
   /* and value strings                                         */

   if  (isCommentLine(tempStr) == FALSE) {
      splitKeywordValName(tempStr,&keywordStr,&valueStr);

      keywordfound = FALSE;
      if (strcmp(keywordStr,"BUFFER") == 0) {
         strcpy(instParms.bufferVal,valueStr);
         sprintf(tempStr,"   Setting BUFFER to %s\n\r",
             valueStr);
         rc = DosWrite(log1FileHandle,tempStr,
            strlen(tempStr),&bytesWritten);
         keywordfound = TRUE;
      }

      if (strcmp(keywordStr,"TIMEOUT") == 0) {
         strcpy(instParms.timeoutVal,valueStr);
         sprintf(tempStr,"   Setting TIMEOUT
            to %s\n\r",valueStr);
         rc = DosWrite(log1FileHandle,tempStr,
     strlen(tempStr),&bytesWritten);
         keywordfound = TRUE;
      }

      if (keywordfound == FALSE) && (strlen(keywordStr) !=0)) {
        sprintf(tempStr,"   Warning:  Invalid
           keyword: %s\n\r",keywordStr);
        rc = DosWrite(log1FileHandle,tempStr,
           strlen(tempStr),&bytesWritten);
        sprintf(tempStr,"      Processing continues\n\r",
            keywordStr);
        rc = DosWrite(log1FileHandle,tempStr,
           strlen(tempStr),&bytesWritten);
      }
   }
}

/* Use default values if the user didn't specify any            */

if (strcmp(instParms.bufferVal,"") == 0) {
   strcpy(tempStr,"   Using default value 100
         for BUFFER.\n\r");
      rc = DosWrite(log1FileHandle,tempStr,
         strlen(tempStr),&bytesWritten);
      strcpy(instParms.bufferVal,"100");
   }
   if (strcmp(instParms.timeoutVal,"") == 0) {
      strcpy(tempStr,"   Using default value 5
         for TIMEOUT.\n\r");
      rc = DosWrite(log1FileHandle,tempStr,
          strlen(tempStr),&bytesWritten);
      strcpy(instParms.timeoutVal,"5");
   }

   /* write out the ini file */
   strcpy(filename,cmdLineParms.targetDir);
   strcat(filename,"CIDEXMP.INI");

   /* delete the old one first                                    */

   rc = DosDelete(filename);

   rc = DosOpen(filename,                /* Name of file to create  */
      &FileHandle,                       /* Address of file handle  */
      &ActionTaken,                      /* Pointer to action       */
      0,                                 /* Size of new file        */
      FILE_NORMAL,                       /* Attribute of new file   */
      FILE_OPEN | OPEN_ACTION_CREATE_IF_NEW,
      OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE,
      NULL);

   if (rc != 0) {
      sprintf(tempStr,"Error:  Could not open INI file %s
         with rc = %ld\n\r", filename,rc);
      rc = DosWrite(log1FileHandle,tempStr,
         strlen(tempStr),&bytesWritten);
      rc = DosClose(log1FileHandle),
      free(rspFileBuffer);
      exit(RC_IOERROR);
      return;
   }

   /* Write out the INI lines                                     */

   strcpy(lineStr,"*** CID EXAMPLE INI FILE  ***");
   strcat(lineStr,"\n\r");
   rc = DosWrite(FileHandle, lineStr,strlen(lineStr),
      &bytesWritten);

   strcpy(lineStr,instParms.bufferStr);
   strcat(lineStr," ");
   strcat(lineStr,instParms.bufferVal);
   strcat(lineStr,"\n\r");
   rc = DosWrite(FileHandle, lineStr,strlen(lineStr),
      &bytesWritten);
   strcpy(lineStr,instParms.timeoutStr);
   strcat(lineStr," ");
   strcat(lineStr,instParms.timeoutVal);
   strcat(lineStr,"\n\r");
   rc = DosWrite(FileHandle, lineStr,
      strlen(lineStr), &bytesWritten);

   rc = DosClose(FileHandle);

   if (rc != 0) {
      sprintf(tempStr,"Error:  Could not close INI file %s
         with rc = %ld\n\r",filename,rc);
      rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
         &bytesWritten);
      rc = DosClose(log1FileHandle);
      free(rspFileBuffer);
      exit(RC_IOERROR);
      return;
   }

  /* Free dynamically allocated memory and close the log file        */

   free(rspFileBuffer);

   DosGetDateTime(&DateTime);
   sprintf(tempStr,"%2.2d-%2.2d-%2.2d %2.2d:%2.2d:%2.2d
      - Completed installation for node %s.\n\r",
      DateTime.month,DateTime.day,DateTime.year,DateTime.hours,
      DateTime.minutes,DateTime.seconds,cmdLineParms.nodename);
   rc = DosWrite(log1FileHandle,tempStr,strlen(tempStr),
      &bytesWritten);

   rc = DosClose(log1FileHandle);
   if (rc != 0) {
      printf("Error:  Could not close log file %s
         with rc = %ld\n",cmdLineParms.log1,rc);
      exit(RC_IOERROR);
   }

   exit(RC_SUCCESS);
}

/*******************************************************************/
/* stringcpyFromTo                                                 */
/* - Copy a string between the from and to ranges.                 */
/*******************************************************************/

int stringcpyFromTo(PSZ source, PSZ target, int from, int to)
{
   int i,j,k,len;

   i = j = k = 0;
   strcpy(target,"");
   if (to < from)
```

```
      return(-1);
   len = strlen(source);
   while (i < len) {
      if ((i >= from) && (i <= to)) {
         target[j++] = source[i++];
      } else {
         i++;
      }
   }
   target[j] = '\0';
   return(0);
}

/*****************************************************************/
/* isCommentLine                                                 */
/* - Determine if the input line is a comment                    */
/*****************************************************************/

BOOL isCommentLine(UCHAR *source)
{
   int i = 0;

   /* Consider blank lines comments to ignore them.              */

   if ((strlen(source) == 0) |  ((strcmp(source,"\r") == 0) |
         (strcmp(source,"\n") == 0))) {
      return(TRUE);
   }

   i = 0;
   while ( (source[i] == ' ') && (i <= strlen(source)) ) {
      i++;
   }

   /* If a line begins with a ; or *, it's a comment line        */

   if ((source[i] == ';') || (source[i] == '*')) {
      return(TRUE);
   } else {
      return(FALSE);
   }                 /* endif                                    */
}

/*****************************************************************/
/* splitKeywordValName                                           */
/* - Split the input string into keyword and value names         */
/*****************************************************************/

int splitKeywordValName(UCHAR *inputStr, UCHAR *keywordStr,
UCHAR *valueStr)
{
   PSZ *tempStr;
   int chr = '=';
   int i,j,k;
   i = j = k = 0;
   tempStr = strrchr(inputStr,chr);
   if (tempStr == NULL) {
      strcpy(keywordStr,"");
      strcpy(valueStr,"");
      return(0);
   } /* endif */
   j = strlen(inputStr) - strlen(tempStr) - 1;
   for (i=0;i<=j;i++) {
      keywordStr[i] = inputStr[i];
   } /* endfor */
   keywordStr[i] = '\0';
   j = strlen(inputStr);

   i++;  /* skip over the equals sign */
   for (i;i<=j;i++ ) {
      valueStr[k++] = inputStr[i];
   } /* endfor */
   trimBlanks(keywordStr);
   trimBlanks(valueStr);
   return(0);
}

/********************************************************************/
/* trimBlanks                                                       */
/* - Remove blanks from the input string                            */
/********************************************************************/

void trimBlanks(CHAR *inputStr)
{
   CHAR tempStr[300];
   int i,j,max;

   i = j = 0;
   strcpy(tempStr,"");
   max = strlen(inputStr);

   while (((inputStr[i] == ' ') |
            ((inputStr[i] == '\r') | (inputStr[i] == '\n')))
            && (i <= max)) {
      i++;
   }
   while ((inputStr[i] != ' ') && (i <= max)) {
      tempStr[j++] = inputStr[i++];
   }
   tempStr[j] ='\0';
   strcpy(inputStr,tempStr);
   return;
}
```

Figure 16-3. *Sample CID installation program.*

```
; Sample CID response file for NODE0001
BUFFER = 50
* Provide a little more time
TIMEOUT = 10
```

Figure 16-4. *Sample installation response file NODE0001.LOG.*

```
12-15-1993 12:36:31 - Started installation for node NODE0001.
   Source Directory - Y:\CID\IMG\PROD\
   Client Filename - NODE0001.RSP
   Group Directory -
   Error Log Filename (L1) - NODE0001.LOG
   History Log Filename (L2) - NODE0001.LOG
   Target Directory -
   CONFIG.SYS Directory - C:\
   Nodename - NODE0001
   Setting BUFFER to 50
   Setting TIMEOUT to 10
12-15-1993 12:36:32 - Completed installation for node
   NODE0001.
```

Figure 16-5. *Generated installation log file NODE0001.LOG.*

SOME SDM PRODUCTS

Although a CID-enabled application need not be concerned with the SDM that a customer is using, it may be interesting for you to have a stratospheric view of some of the SDM products that are available and how they are used.

The LAN CID Utility (LCU) is one such SDM that is packaged as part of the IBM Network Transport Services/2 (NTS/2) product. LCU uses SRVIFS, which allows a client workstation to access files on a remote server workstation on a LAN using the NETBIOS protocol. (SRVIFS and NETBIOS are included with NTS/2.)

LCU files are REXX command files that issue installation commands, re-IPL machines when needed, track the installation state of the target machine, and allow the ordering of product installation (for example, OS/2 gets installed before IBM LAN Services 3.0).

LCU does not require the user to be involved at the workstation other than to initiate the process from a boot diskette at the client machine. Thus, it provides for lightly attended installation.

Another product, the IBM NetView Distribution Manager/2 (NetView DM/2), also supports CID-enabled applications, but it uses change file install descriptions with the installation details for a product that can be scheduled on a workstation. NetView DM/2 requires more resources than LCU, but it provides complete unattended installation and more flexibility in how applications can be installed and maintained.

TIPS

The following lists tips that you can use when designing your CID-enabled application:

- ☐ When there are choices, make sure you have documented for the user exactly how your installation program will treat the choice. Will your program use a default value when a keyword is not included in the response file, or will your program will log an error and terminate?

- ☐ Keep your response file structure simple, if possible, and make the keywords understandable. This will make life easier on the already overburdened user and LAN administrator.

- ☐ Supply a default .RSP file for users to utilize as a template in creating their own. Pepper it with comments too.

- ☐ Include a command-line parameter or a function of your program to create a .RSP file based on the current product configuration on the workstation on which the application is running. This allows the user to easily replicate a standard configuration across multiple machines without having to investigate and create a response file from scratch. Curious users can also utilize this feature to examine the configuration of their machines in an ASCII file.

- ☐ Have your installation program insure that enough space exists for your application on the target drive of the client workstation before starting the installation process. Doing this will prevent your program from needing to clean up after running out of disk space.

- ☐ Be sure to delete any temporary files or directories that your installation program may have created. This includes any unnecessary compacted files that may have been copied to the client workstation as part of the installation process.

- ❑ Update your error, listing, and other logs in real time rather than after the installation or configuration process is complete. This will give the administrator an accurate record of the activities in case the installation or configuration sequence terminates abnormally. (A large boulder could tumble down onto an administrator's LAN domain controller, for example.)

- ❑ Make it easy for administrators to remove your product or install over a previous version of your product on a client workstation. This can be built into your installation program too.

- ❑ Use HPFS instead of FAT partitions to improve the performance of your code server. The SDM will be able to transfer data faster from an HPFS partition.

- ❑ Keep your disk images or product files in compressed form on the code server. This will reduce the disk space requirements and shorten the transmission time of the files across the LAN.

SUMMARY

With the explosive growth in LANs, users are demanding an easier way to configure, install, and distribute their applications across all the workstations on their LAN. The CID strategy provides an easy way in which to enable applications to participate in the LAN environment. You may find it odd that many of the features of making your product CID enabled are presented as recommendations rather than requirements.

CID was designed to allow application developers, such as yourself, as much leeway as possible in designing their products, while still being able to participate in the CID environment. By including these suggested functions in your design, you can make your program more attractive to network administrators and users, as well as making it easier to learn and use.

CHAPTER 17

Object-Oriented Programming

"One doesn't discover new lands without consenting to lose sight of the shore for a very long time." —*André Gide*

INTRODUCTION

Object-oriented programming and design is the latest paradigm in software development. It is a way of modeling a system that closely parallels the way you think of the real world system. This model helps you reduce the complexity of your programs, making them easier to design, code, and debug, which in turn reduces the time you need to get your product to market.

The object-oriented paradigm can also be applied to the user interface to make your programs easier to learn and use, giving you an important edge in today's competitive market.

Object-oriented programming requires that you first decide what types of objects you need to model and then what you can do with each type. Similarly, an object-oriented interface has the user first select an object, such as a folder, and then decide what to do with it.

In this chapter you will learn the concepts and terminology associated with object-oriented programming. The chapter concludes with an example of designing an object-oriented class and subclass.

The chapters that follow on System Object Model (SOM) will show you how to implement these and several other classes.

THINKING A DIFFERENT WAY

The object-oriented programming paradigm is a natural follow-up to the way software development has changed and grown over the years. Originally programmers had to think in terms of zeroes and ones, since the program was literally a string of binary bits.

Assembly languages went one step further, allowing programmers to think in terms of registers, bytes, and words. High level languages freed us from having to know the architecture of a machine and opened the way for many different types of design, including procedural, functional, and logical.

In a procedural programming language you model functions and data and then explicitly tell the computer exactly what sequence of steps to take. The flowchart was a good tool to use, since it modelled the sequence of steps from function to function. C is a good example of a procedural programming language.

In a functional programming language, you still model functions, but now the data is part of the function as well. Some functional languages added constructs for defining data. Lisp is a good example of a functional programming language.

In logic (or rule-based) programming, unlike either procedural languages or functional languages, you do not tell the computer what steps to take. Instead, you specify a collection of facts and a collection of rules for making valid deductions. A new fact is true if it can be derived from the existing facts and rules. The sequence of steps needed to prove or disprove facts is left completely up to the language's engine. Prolog is a good example of a rule-based programming language.

The object-oriented approach is different from all three of these. Objects are the focus, rather than facts, functions, or data. Data and functions are defined as part of the object itself, rather than as separate entities. The data and functions are actually bound into one package along with the objects they support or act on.

The functions can be as general or as specific as you want. To invoke a function on a particular object, the caller sends a message to that object, telling it which function to execute and giving it any necessary supplemental data. The caller can be an object as well; objects often invoke functions on other objects.

OBJECT-ORIENTED CONCEPTS AND TERMINOLOGY

Object-oriented programming has a number of key concepts you will need to learn, as well as a few new terms. These are discussed in the following sections:

- ❑ Objects and Data
- ❑ Methods and Messages
- ❑ Classes
- ❑ Subclasses, Class Hierarchies, and Inheritance
- ❑ Encapsulation
- ❑ Polymorphism
- ❑ Metaclasses

Objects and Data

An *object* is any entity you want to model. For example, if you were writing an application for a library, you would call a book an object. An object consists of *data* and *methods*. Data for a book might include the title, the author, the number of pages, the current status (whether or not the book is checked out of the library), and the text in the book.

Objects can consist in part or in whole of other objects. Suppose you also wanted to model pages. Then instead of the data item containing the text, a book object would have a data item containing a list of page objects.

Methods and Messages

Methods represent all the things you can do with or to the object. Methods for a book might include Open_the_book, Turn_the_page, and Close_the_book. When you want to do something with an object, you must invoke a method on it. You invoke a method on a particular object by sending a *message* to the object telling it which method to perform, and including any extra data that that method may need. The concept of sending messages is to allow an object-oriented programming language the freedom to decide how to implement method invocation.

Classes

Similar objects are grouped into *classes*. A class is an abstract template containing descriptions of what data and methods are associated with objects of that class. Given an object's class, you will not be able to uniquely identify the object, but you will know what kinds of things you can do with it.

For the examples in the preceding paragraphs, you would define a book class with five data elements and three methods. To illustrate the idea of objects within objects, you would need to define a page class as well.

An object is called an *instantiation* of, or instance of, a class. Each object in a class has the same types of data but may have different values for that data. For example, every book has a title, but not all books have the same title.

- ❑ *Calculator Example:* Suppose Company A makes a calculator that can only add and subtract. You would define a calculator class with methods Add and Subtract. Its data would be Register1 and Register2. Every object of the calculator class contains two data elements, named Register1 and Register2, and every object of the calculator class has two methods, named Add and Subtract. The actual instance data will vary from calculator object to calculator object.

Subclasses, Class Hierarchies, and Inheritance

A *subclass* is a class that is derived from another class. The subclass is said to be a *child* of the class it was derived from, and the original class is said to be the *parent* of the new subclass. A subclass is a specialization of its parent. For example, employees are special human beings. Anything a human being can do, an employee can do, and any characteristics of a human being are also characteristics of an employee.

The terminology follows logically from the family relationships: children, parents, grandparents, ancestors, and descendants. A class hierarchy is an extended family, consisting of the set of classes obtained by starting at one class and including all ancestors and descendants of that class.

Inheritance is one of the key concepts in object-oriented programming. A subclass automatically *inherits* all of the data and methods of all of its ancestors. In other words, any method that can be invoked on a class can also be invoked on any of its subclasses. A datum or method that comes from an ancestor is said to be inherited. You do not have to change the code of the parent class in any way when you define a subclass of it.

Inheritance is the main vehicle that allows code to be reused. From a design point of view, you can use all of the inherited data and methods exactly as if they had been first defined on your subclass itself, rather than on an ancestor. In practice, however, this will vary from language to language.

- ❑ *Calculator Example Revisited:* Suppose Company A makes a new calculator that can multiply and divide as well as add and subtract. This is a superset of the existing calculator class, since it does everything the existing one does. You would define a subclass of the calculator class called newcalc. The newcalc class will inherit all of the attributes of the calculator class, meaning it has two registers and two methods. You now have only to define two new methods on the newcalc class, one for multiplying and one for dividing.

Subclassing has many advantages and is appropriate in many circumstances. You see how it can save you from writing a lot of code. You can also subclass when you need to do just a little bit more than the parent method. You can use it as a logical arrangement as well, to give some structure to the classes you are working with.

Subclassing and inheritance gives you the ability to move methods from class to class within a hierarchy. If you move a method from a child to a parent, everything should still work.

Descendants will still inherit the method since they are lower in the hierarchy, so they should not need to recompile. Siblings who now inherit the method don't use it, so they should not need to recompile either.

In practice, the need to recompile will be based in part on the particular language implementation. An implementation that meets all the design points of object-oriented programming will not need to recompile siblings.

Suppose engineers at Company A decide to make a less expensive calculator, perhaps one that can add, subtract, and multiply, but not divide. The engineers will not have to write any new code. They can define a third calculator class and place it into the hierarchy between the calc and newcalc classes. Then they can move the Multiply method up from newcalc into the third class. Voilá!

The original calculator can only add and subtract. The third calculator (in the middle of the hierarchy) inherits the add and subtract methods and defines the multiply method. The newcalc class inherits all three of those methods and still defines the divide method. Figure 17-1 shows the initial two-calculator hierarchy and then the new three-calculator hierarchy.

INITIAL HIERARCHY

CLASS	PARENT	METHODS
calc	--	Add, Subtract
newcalc	calc	Add(I), Subtract(I), Multiply, Divide

NEW HIERARCHY

CLASS	PARENT	METHODS
calc	--	Add, Subtract
newcalc2	calc	Add(I), Subtract(I), Multiply
newcalc	newcalc2	Add(I), Subtract(I), Multiply(I), Divide

Note: A method followed by (I) is one that is inherited from an ancestor.

Figure 17-1. *Modifying a class hierarchy.*

Subclassing and inheritance are powerful tools in object-oriented programming.

Encapsulation

A second key concept in object-oriented programming is called *encapsulation*. This concept allows you to hide details of a class from anyone outside the class itself. This reduces the complexity of the code by only surfacing details that need to be seen. It also prevents errors by isolating parts of the implementation. Encapsulation is enforced by any true object-oriented language.

No one can cheat and manipulate your internal structures directly; they must go through your methods. This allows you to change your implementation without causing any breaks elsewhere in the system. Encapsulation also forces a discipline on the software in that no implicit side affects are allowed. An object cannot change any data or objects that are not passed to it.

You will often see words such as *public* and *private* when reading about object-oriented programming. *Exposed* and *hidden* are also used. The exact meaning will vary from language to language, but the common idea is that anything that is public, or exposed, is accessible to anyone having access to the class. Private, or hidden, indicates some level of encapsulation. Three different flavors of privacy are the following:

- ❑ Completely hidden from anyone outside the class, including descendants
- ❑ Hidden from anyone outside the class, but accessible by descendants
- ❑ Accessible by anyone whom the class designates as all right

Polymorphism

A third key concept in object oriented programming is called *polymorphism*. This is similar to operator overloading, where an operator can work on different kinds of input.

For example, the multiplication operator can be used with integer expressions or with scientific notation expressions, even though the implementation will be different for each type of expression.

Polymorphism is very useful when a function name should be used by many different types of object. In most nonobject-oriented languages you would have to have a different name for each type, since only the built-in functions can be overloaded.

The reason polymorphism is key to object-oriented programming is that every method is bound to a particular class. An Add method for a calculator object would return the sum of two numbers, whereas an Add method for a list would make the list longer.

There is no collision over which Add method to call—the method is invoked on an object. That object is an instance of some class, and that class has only one method named Add.

You can see that object-oriented designs are more natural than the usual top-down or bottom-up designs. They make the overall system design less complex, making it easier to code and maintain.

Object-oriented designs are extremely modular, making them easier to modify. They encourage code reuse through inheritance. Finally, polymorphism simplifies your task as a programmer by allowing you to name things in a natural way.

Metaclasses

A *metaclass* is the class of a class. It is a class in its own right, whose data and methods are associated with a class as a whole, rather than with an object of a class. A metaclass might have a data member containing the number of instantiated objects of the class and a method for creating objects of the class. A metaclass is an overseer for a class.

As an example, define an automobile class. It will have data items such as color, year, and number of doors, and methods such as shift, start, and brake. How is a new car created? There is an autombile manufacturer. When you want a new car made, you ask the manufacturer. If you want to know how many cars were made in a year, you ask the manufacturer. If you want to know if there are any blue cars available, you ask the manufacturer. All of these questions are questions about the class of cars as a whole, as opposed to questions about a particular car.

These kinds of questions, and the data and methods required to answer them, are what makes a metaclass the class of a class. In this example, manufacturer is the metaclass for the automobile class. Now consider the manufacturer.

The manufacturer has data and methods of its own to help it keep track of all the automobiles. It would have methods such as makeNewCar and data such as a list of years. Every object has a class, and manufacturer has all the elements needed to be modelled as an object. The result is that you have an automobile class and a manufacturer class, and manufacturer is the metaclass of automobile.

DESIGNING AN OBJECT-ORIENTED CLASS HIERARCHY

Let's illustrate the approach and advantages to objects. Consider a FIFO queue of homogeneous elements. You find these in many places, from lines at the bank to lines at the movies to cars stopped at a red light. The goal of this discussion is to model a FIFO queue.

- ❑ The first step is to define a class named FIFOQ. Every queue object will be an an instantiation of the FIFOQ class.
- ❑ The next step is to define the data and methods needed for a FIFOQ class, since object-oriented design combines both data and methods.

The obvious methods are add and get, and some kind of structure must be maintained to handle the data in the queue. The FIFOQ class will have two methods, AddItem and GetItem, and four data items, which are pListHead, pListTail, bListEmptyFlag, and usListItemSize. Since pListHead and pListTail are not needed by callers of the FIFOQ class, they will be made private. Figure 17-2 shows the public part of this class definition, in pseudocode.

```
Class:  FIFOQ
Methods:        INT     RC = AddItem(PVOID *pItem)
                PVOID   Item = GetItem()
Data:           USHORT  usListItemSize
                BOOL    bListEmptyFlag
```

Figure 17-2. *Public methods and data in a queue class.*

A particular queue, or queue object, is an instantiation of the FIFOQ class. It shares the same code as any other queue object, but has a unique set of data. This is similar to how shared code such as editors work—the code is the same but all users get their own data.

Anyone wanting to manipulate a queue would use one of the two methods defined on it. To invoke a method on a queue the caller would send a message to the particular queue, indicating which method to perform as well as any supplemental information needed, if any. For example, you would need to pass the item to add for the AddItem method.

The implementation details should be totally hidden from the user, so the methods and data needed to manipulate the queue are not exposed. In this case, all of the methods and two of the data items should be exposed, and two data items should be hidden. Figure 17-3 shows the hidden description for the FIFOQ class, in no particular language. A user should be able to write code that uses queue objects without seeing anything about the queue class other than the information in Section 1. Figure 17-3 shows C code for the methods. (Chapter 18 shows a complete SOM C language implementation of this queue class.)

```
SECTION 1:  EXPOSED
Class:  FIFOQ
Methods:    INT     RC = AddItem(PVOID *pItem)
            PVOID   Item = GetItem()
Data:       USHORT  usListItemSize
            BOOL    bListEmptyFlag

SECTION 2:  HIDDEN
Data:     typedef struct ListItem {
                          struct ListItem  *pNext;
                          struct ListItem  *pPrev;
                          UCHAR            cData;
                     } ListItem, *pListItem;

          pListItem  pListHead;
          pListItem  pListTail;
```

Figure 17-3. *Methods and data in a queue class.*

```
typedef struct aList {
     pListItem    pListHead;
     pListItem    pListTail;
     USHORT       usListItemSize;
     BOOL         bListEmptyFlag, public;
        } List;

typedef struct ListItem {
             struct ListItem  *pNext;
             struct ListItem  *pPrev;
             UCHAR            cData;
        } ListItem, *pListItem;

INT    AddItem(PVOID  pListIn,  PVOID  pItem)
{
    pListItem   pNewItem;
    INT         rc = 0;
```

```
   List        *myList;                    /* For convenience only */

   do {
      myList = (List *)pListIn;

         /* Allocate space for a new ListItem                          */
      pNewItem = malloc(sizeof(ListItem)-sizeof(UCHAR) +
                        myList->usListItemSize);
      if (!pNewItem) {
         rc = MEMORY_ERROR;
         break;
      }
         /* Initialize the new Item                                    */
      memcpy(&pNewItem->cData, pItem,
                       myList->usListItemSize);
      pNewItem->pNext = (pListItem) NULL;
      pNewItem->pPrev = myList->pListTail;

         /* Add the new Item to the end of the List                    */
      if (myList->bListEmptyFlag == LIST_IS_EMPTY) {
         myList->pListHead = myList->pListTail = pNewItem;
         myList->bListEmptyFlag = LIST_NOT_EMPTY;
      } else {
         myList->pListTail->pNext = pNewItem;
         myList->pListTail = pNewItem;
      }

   } while (0);              /* enddo                                  */

   return rc;
}

/********************************************************************/

PVOID  GetItem(PVOID  pListin)
{
   pListItem   pSaveItem;
   List        *myList;                    /* For convenience only */

   PVOID       retItem = NULL;

   do {
      myList = (List *)pListIn;

      /* Error if the list is empty                                    */
      if (myList->bListEmptyFlag == LIST_IS_EMPTY) {
         break;
      }


      /* Copy the item from head of the chain                          */
      /* Allocate return buffer                                        */
      retItem = (PVOID) malloc( myList->usListItemSize);
      if (!retItem) {
            break;
         }
         memcpy(retItem, &myList->pListHead->cData,
                               myList->usListItemSize);
         pSaveItem = pListHead;


         /* Remove the item from the list                                */
         /* Cases: 1 item in the list                                    */
         /*        > 1 item in the list                                  */
         if (MyList->pListHead == myList->pListTail) {
            /* Only one item in the list                                 */
            myList->pListHead = myList->pListTail =
                      (pListItem) NULL;
            myList->bListEmptyFlag = LIST_IS_EMPTY;
         } else {
            myList->pListHead = myList->pListHead->pNext;
            myList->pListHead->pPrev = (pListItem) NULL;
         }

         /* Now free the list item itself                                */
         free(pSaveItem);

      } while (0 );          /* enddo                                    */

      return retItem;
}
```

Figure 17-4. *C code for list class.*

Metaclass for the Queue Hierarchy

Now, you need some way to create three queue objects. To do this, a CreateQueue method is required. The CreateQueue method cannot be invoked on the queue itself because the queue does not exist yet. There are other functions about queue objects that you might want. For example, how many queues have been instantiated? How much memory is used in the system for queue items?

As mentioned earlier, a class whose data and methods are associated with another class, rather than with an object, is called a metaclass. CreateQueue is a method in a metaclass. NumLists is a data item in a metaclass. The only difference between a class and a metaclass is the type of methods it contains.

Figure 17-5 shows the definition of the MetaQueue class, and Figure 17-6 shows the C implementation of the CreateQueue method.

```
SECTION 1:  EXPOSED
Class:  MetaQueue
Methods:  PVOID = CreateQueue(USHORT usDataSize)
          void    DestroyQueue (PVOID Queue)

SECTION 2:  HIDDEN
Data:     USHORT     usNumQueues= 0;          /* Global or static var*/
```

Figure 17-5. *Methods and data in the MetaQueue class.*

```
static USHORT numQueues = 0;

PVOID  CreateQueue(USHORT  usDataSize)
{
   List   *pList;

      /* Allocate space for a new Queue                                */
   pList = malloc(sizeof(List));
      /* Initialize the data                                           */
   pList->pListHead = pList->pListTail = (pListItem) NULL;
   pList->usListItemSize = usDataSize;
   pList->bListEmptyFlag = LIST_IS_EMPTY;
      /* Increment the total number of lists                           */
      /* Note that this variable would have to be                      */
      /* decremented in DestroyQueue                                   */
   usNumQueues++;

   return (PVOID) pList;
}
```

Figure 17-6. *C code for the CreateQueue method.*

Subclasses in the Queue Hierarchy

There are many different kinds of queues. Consider FIFO and LIFO queues. Most of the manipulations are the same. They differ in that one adds data to the end of the list and the other adds data to the front of the list.

Recall that a subclass is a class that inherits properties from another class, called the parent class. To reuse all of the code in the FIFOQ class, you would define the classes GENQ and LIFOQ, where GENQ was the parent of both FIFOQ and LIFOQ.

Then move the methods from FIFOQ up to GENQ, and modify AddItem to add to the head or the tail based on an input flag. Override AddItem in FIFOQ and LIFOQ to call the parent with the proper flag.

C is not an object-oriented language (you cannot subclass in it), so code for these classes will be shown in the next chapter, which describes the object-oriented engine that comes with OS/2 2.1.

SUMMARY

Object-oriented programming and design is a new methodology for modeling systems. It makes your task as a programmer easier by making the design less complex, by making the code easier to debug, modify, and maintain through encouraging and enforcing encapsulation and modularity, by allowing polymorphism, and by increasing code reuse through subclassing and inheritance. It makes the user's task easier by reducing the learning curve and by requiring little or no technical knowledge.

CHAPTER 18

System Object Model (SOM)

"One of the greatest pains to human nature is the pain of a new idea."
—William Bagehot

INTRODUCTION

One of the unique features of OS/2 2.1 is that it comes with built-in support for object-oriented programming. IBM's System Object Model (SOM) is a highly flexible and powerful tool. It is not an entire programming language in itself. It is an engine that supports multiple programming languages. The Workplace Shell uses SOM to support its object-oriented interface.

Among the many advantages of SOM, you will find language independence. Any programming language can be supported by SOM. To use this feature you must have SOM *bindings* for your particular language. Bindings, as you may guess, are the glue between SOM and your language.

Unless you're a compiler writer (or have a friend who is), you will need to choose a language that already has bindings. At the time this book is being written, C and C++ have bindings and more are being prepared. If your favorite language is not among them, write to the company that makes its compiler!

A second advantage of SOM is related to its language independence. One restriction among many of today's object-oriented class libraries is that they can be used only by a class that is written in the same language as the library itself. For example, a C++ class cannot be subclassed in Smalltalk.

SOM lifts this restriction. Any class written to use SOM can be accessed and subclassed by any other class written to use SOM, regardless of what programming languages the classes are actually written in.

Another advantage of SOM is that it is platform-independent and currently available on both OS/2 2.x and AIX. Your applications will not be tied to one operating system unless you make system calls within your program, and even then all you will have to do is port the system calls. This gives you a broader market for your product.

SOM consists of both a development environment and a run-time environment. You specify a class definition, or interface, in the development environment. Here, SOM provides tools for assisting you in defining and modifying classes. The run-time environment provides support for SOM methods, which are used by both the class implementation and the applications that use the class. Notice that applications do not need to use SOM's development environment unless they also define a class.

As a development environment, SOM goes a long way toward making life easier for you. When modifying a program, you often have to make concessions to retain backward compatibility. Otherwise existing applications that use your code have to be recompiled. (Applications that use a class will be called *clients*.) Some modifications are almost inevitable. If you use some of the compatibility functions of SOM, you will be able to make many kinds of modifications to your code without having to recompile existing clients.

For example, you can add new methods without having to modify or even recompile existing clients. You can also add new data (variables) without touching existing clients. You can add, modify, or delete private data, causing subclasses to need recompiling, but again neither clients nor ancestor classes will know the difference. You may decide that a method is needed higher up in the class hierarchy (that is, in a ancestor class). You can even change an entire implementation. Unless the public method definitions change, the existing clients will be compatible with the new implementation.

The rest of this chapter will show you the details of the SOM run-time and development environments and teach you how to use SOM to define classes. It will discuss the macros and conventions that SOM supplies for simplifying method invocation and access to instance variables. It will also show you the methods needed to create and destroy objects. Chapter 19 will cover many of the SOM methods and macros that you will find useful when writing and debugging your class implementations.

FLOW OF FILES

The development process for SOM classes is a little different from that for a regular C program. As a C programmer, you are used to three different kinds of files: C files, header files (H), and DEF files, two of which you code by hand. With SOM, there are five different types of files, one of which you code by hand, one of which you modify, and three of which you never alter:

- ❑ Language-dependent class definition files (CSC)
- ❑ Language-independent class definition files (SC, PSC)
- ❑ Class implementation file (C)
- ❑ Header files (H, IH, PH)
- ❑ DEF file

Interestingly enough, the header files are among those you never alter, and the C file you only modify. At this point, if you are not familiar with the concept of public and private information in object-oriented programming, you probably should go back and read Chapter 17.

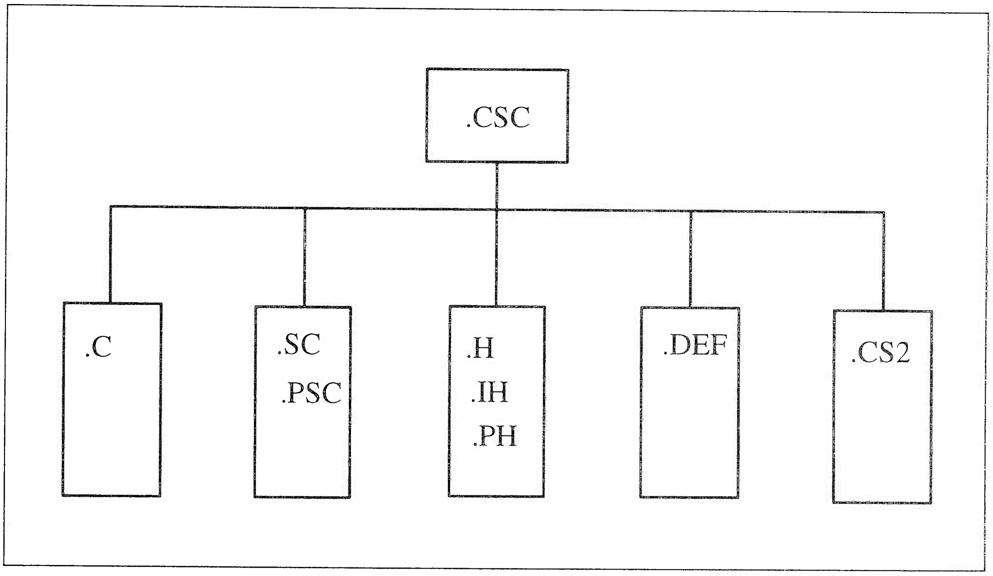
To define a class in SOM, follow these steps:

- ❑ Write a class specification using Object Interface Definition Language (OIDL), a language invented and brought to you by IBM, the creators of SOM. This specification will have some language-dependent terms in it. For the C language, the name of the file containing the OIDL class specification will have the extension .CSC.

- ❑ Run the SOM compiler. The SOM compiler can be thought of as a collection of preprocessors. These preprocessors are called *emitters*, since each one emits a particular type of file. For example, the SC emitter will generate an SC file. The SOM compiler will generate several files for you based on which emitters you have chosen to enable. Figure 18-1 shows the different files generated for a class to be implemented in C. The CSC file is discussed later in *The OIDL Language—Basic CSC Syntax*. The files generated by emitters are discussed later in *An OIDL Class Specification*. Which emitters to choose is discussed in *The SOM Compiler Environment Variables*, and an example is given in Chapter 19 in *Compiling and Linking a SOM Class*.

- ❑ Add code to the C file (the class implementation file; could be C++ as well, or others in the future)
- ❑ Run the C compiler (C Set++ Version 2.0, or C/Set2).
- ❑ Is the implementation satisfactory? If not, go back and modify it. If you change the CSC file, you must repeat the steps starting from running the SOM compiler. If you change only the C code, you must repeat the steps starting from running the C compiler.
- ❑ Run the linker and build a DLL, an EXE, or both.

Figure 18-1 below and its accompanying table on the next page explain the SOM files that will be implemented for C.



EXTENSION	MODIFIABLE BY	PURPOSE
.CSC	Programmer	Language-dependent class specification
.SC	SOM	Public language-independent class specification
.PSC	SOM	Private language-independent class specification
.IH	SOM, Passthru	Header file for the class implementor only
.H	SOM, Passthru	Header file for users of the class
.PH	SOM, Passthru	Header file for private users of the class
.DEF	Programmer, SOM	DEF files for DLLs
.CS2	SOM	Formatted version of the CSC

Figure 18-1. *SOM files for a class to be implemented in C.*

The CSC file has the C language OIDL specification for a class. When you run it through the SOM compiler, the other files are generated for you, depending on which emitters you have enabled. The C file has a template for you to fill to implement your class's functions. It is discussed later in this chapter in *The SOM-Generated C Template*, and it is strongly recommended that you use it. The SC and PSC files contain the language-independent specifications for your class. Programmers wanting to subclass your class would include the SC file, even if they are writing the subclass in the same language you used for your class. If you want to allow them access to the class's private interface, you would give them the PSC file and they would include that instead. (The PSC file contains an include statement for the SC file.)

The H, IH, and PH files are header files. The IH file (implementation header) should be included only by the class implementation file (the C file). The H file is included by anyone wanting to use the public interface to your class. As with the PSC file, the PH file is included by anyone wanting to use your class whom you trust to have access to the class's private interface.

The DEF file is used the same way a non-SOM generated DEF file is used. Chapter 19 covers its use when building your SOM class into a DLL. If you are using a language other than C, then whoever wrote the emitters for it must tell you what files can be generated.

THE SOM RUNTIME

SOM has both a runtime component and a development component. The SOM runtime must be available in any process that uses a SOM-defined class. It provides dynamic creation of objects, dynamic method name resolution when necessary, and dynamic dispatching of methods (message passing, in generic object-oriented terms). It also provides three classes that are the ancestors of all SOM-defined classes, including those you define and implement yourself.

SOM provides three classes that you must use when defining your own classes: SOMObject, SOMClass, and SOMClassMgr. These classes include all of the SOM-supplied methods that support Object-Oriented programming. You will not need to know all of the SOM supplied methods. The fact that your objects inherit methods by being descended from one of the three classes is enough to allow SOM to invoke them for you.

SOMObject

All SOM objects, whether an instance of a class or an instance of a metaclass, must be descended from SOMObject. The SOMObject class is the only SOM class that does not have a parent. Its metaclass is SOMClass, described in the next paragraph. The SOMObject class contains many methods which your objects inherit and use, either explicitly or implicitly. These include methods such as somIsA, somInit, somFree, and somGetClassName (these methods are covered in detail in Chapter 19).

SOMClass

All SOM class objects must be descended from the SOMClass class. Following the rule for SOMObject, the SOMClass is descended from SOMObject. And following the rule for SOMClass, the SOMClass's metaclass is SOMClass. It is the only class whose metaclass is itself. The SOMClass class contains many methods which class objects use, such as somNew and somInitClass. It also inherits all of the methods from SOMObject, including somInit, somFree, and so on.

SOMClassMgr

The SOMClassMgr class is like a database server. The SOMClassMgr instance maintains information about all of the classes that are known within the process. It coordinates the class, class object, instances of the class, and location of data about the class (such as what file it is in). The SOMClassMgr is descended from SOMObject, and its metaclass is SOMClass. It contains methods such as somFindClass and somLocateClassFile. One and only one instance of this class is created in the SOM runtime.

When the SOM runtime is initialized, four objects are created. Three are class objects, one for each of the three classes provided by SOM: a SOMObject class object, a SOMClass class object, and a SOMClassMgr class object. The fourth object created is an instance of the SOMClassMgr class.

Table 18-1 shows the relationships among the classes provided by the SOM runtime. Table 18-2 shows the relationships among the four objects created during runtime initialization.

Class	**Parent**	**Metaclass**
SOMObject	none	SOMClass
SOMClass	SOMObject	SOMClass
SOMClassMgd	SOMObject	SOMClass

Table 18-1. *SOM-supplied classes.*

Object	**ClassObject**	**InstanceOf**
SomObject class	Yes	SOMClass
SOMClass class	Yes	SOMClass
SOMClassMgr class	Yes	SOMClass
SOMClassMGR instance	No	SOMClassMgr

Table 18-2. *Objects created during initialization of the SOM runtime.*

If Table 18-2 seems confusing, recall that a class object is an instance of the class's metaclass, not an instance of the class itself. Since SOMObject's metaclass is SOMClass (see Table 18-1), the SOMObject class object will be an instance of the SOMClass class.

THE SOM COMPILER—ENVIRONMENT VARIABLES

There are two aspects to the SOM compiler that you need to know about. You need to know the syntax and options available, and you need to know how to set up your environment to run it.

It will help if you think of the SOM compiler as a precompiler, or set of precompilers. It is actually a precompiler with a set of emitters that the precompiler may invoke.

The SOM compiler uses three environment variables:

- ❑ SMEMIT
- ❑ SMINCLUDE
- ❑ SMTMP

The SMEMIT environment variable is a list of which emitters to run, separated by semicolons (;). For the C language, the name of the emitter is the extension of the file it generates. For example, to emit all of the files typically needed for an EXE you would specify:

```
        set SMEMIT=c;ih;h;ph;sc;psc
```

If the SMEMIT environment variable does not contain any emitters, the SOM compiler will check the OIDL syntax but will not generate any files.

If a C file already exists, it will not be erased. Rather, the SOM compiler will add new templates to the end of the C file if it finds any new methods declared in the CSC file. It will not change existing templates or code. This feature is useful, since you will make extensive changes to the C file when you fill in the method templates.

If SOM overwrote them, you would lose all that work. On the other hand, it means you must be careful when modifying the definition of an existing method (altering the CSC file), since SOM will merely add a new method template to the C file.

The header files and language-independent class specifications will be erased and completely rewritten, so do not modify them by hand. Instead, anything you want to add to a header file you should put in a Passthru section in the CSC file.

The SMINCLUDE environment variable is a list of directories to search for SC files. The syntax is identical to that of the PATH environment variable: a list of directories, either absolute or relative, separated by semicolons (;). The compiler will need to find the SC file of every ancestor of your class, so be sure to specify them. Don't forget that the SOMObject class (somobj.sc) is among them.

For example,

```
set SMINCLUDE=c:\toolkt21\sc;.;
```

tells the SOM compiler to look for SC files in the OS/2 2.1 toolkt21\sc directory and in the current directory.

The SMTMP environment variable specifies what directory the SOM compiler should use for temporary, intermediate files. This directory should be different from the ones that the SOM compiler uses for input or output. For example,

```
set SMTMP=c:\tmp;
```

SMTMP defaults to the root directory of the current drive.

THE SOM COMPILER—SYNTAX

This section on syntax uses four special characters:

[ ] Items enclosed in square brackets are optional.

< > Items enclosed in diagonal brackets are user-supplied values.

As in the *OS/2 Technical Library SOM Guide and Reference*, constants are printed in **bold**, and user-supplied values are printed in *italics*.

The syntax to invoke the SOM compiler is the following:

sc [-<*options*>] *file*[.**csc**]

The extension for the filename is assumed to be .CSC. This can be overridden by specifying an extension or by specifying the -i option.

You can specify any number of options, or you can take the defaults. If you are not using the IBM C SET++ compiler, you must specify the -cl386 option. You can specify the options in any order, either in a group or individually.

The only restriction is that if an option has a parameter, such as the cstyle option, you must specify it as either the last in the group or individually.

The options are the following:

-C <*n*>	Sets the size of the comment buffer. Default = 32K.
-S <*n*>	Sets the size of the buffer for all of the names and passthru lines. Default = 32K.
-V	Displays version information for the SOM compiler.
-d <*dir*>	Places emitted files in this directory.
-h, **-?**	Shows help information for the SOM compiler.
-i <*file*>	Uses the filename as is. Does not add the .CSC extention.
-r	Ensures that all of the entries in the Release Order section actually exist. Default = false.
-**s** <*string*>	Pretends that SMEMIT is set to string. Use this to override the environment variable without actually changing it. The value "string" must be enclosed in quotes, for example, sc -s "c;h;ih" foo.

-**a** <*name*[**=**<*value*>]>	Adds a global attribute. The global attributes available with SOM 1.0 are the following:

comment=<*commentstring*>	where commentstring is "*", or "\\". Tells the SOM compiler to ignore any comments in the specified form and not put them in any of the emitted files. Comments beginning with "#" are always ignored unless they are in a passthru section. Comments in the passthru sections are always emitted. No default.
cstyle=<*commentstyle*>	where commentstyle is s, c, or +. Tells the SOM compiler to generate comments in the indicated form. Specify s for --, c for /**/, and + for //. Default is /**/.
ibmc	Tells the C language emitters (EMITC, EMITH, EMITIH, EMITPH, and EMITDEF) to generate pragmas specifically for the IBM C SET2 ++ compiler.
cl386	Tells the C language emitters (EMITC, EMITH, EMITIH, EMITPH, and EMITDEF) not to generate pragmas specifically for the IBM C SET2 ++ compiler.
r	Ensures that entries in the Release Order section actually exist.
s <*string*>	Ignores SMEMIT and uses string instead.

Table 18-3 summarizes the SOM compiler options.

OPTION	VALUE	PURPOSE
C	<*n*>	Set the size of comment buffer.
S	<*n*>	Set the size of buffer for names and passthru files.
V		Display version number.
d	<*dir*>	Redirect output files.
h		Display help information.
?		Display help information.

OPTION	VALUE	PURPOSE
i	*<file>*	Do not add default extension to the filename.
r		Check that entries exist in Release Order.
s	*<string>*	Ignore SMEMIT; use string.
a	*<name*[=*<value>*]	Add a global attribute.

Table 18-3. *SOM compiler options.*

THE OIDL LANGUAGE—SYNTAX FOR COMMENTS

There are three distinct types of comments in SOM:

- ❑ The first is the one you are used to—the comments you write to describe your program.
- ❑ The second is similar to the first except that SOM writes them. When you run the SOM compiler, it generates comments and places them in the emitted files.

  These comments include information such as the name of the file and the version of the emitter that produced it.
- ❑ The third is comments you write to describe the OIDL. These are called *throwaway* comments since they are not placed in any of the emitted files.

To give you maximum flexibility, the SOM compiler supports four different styles of comment for C language specifications. These are summarized in Table 18-4, with the notes continuing on the next page.

STYLE	EXAMPLE	NOTES
/* */	/* This is a comment */	(1)
--	-- This is a comment	(1)
//	// This is a comment	(1)
#	# This is a comment	(2), (3)

```
Notes:
(1)   The style for SOM-generated comments can be set
      using the -a cstyle=<x> SOM compiler option.
(2)   Throwaway comment; will not be put into any
      emitted files unless it is in a Passthru
      section.
(3)   One additional style can be designated as
      throwaway comments using the -a
      comment=<string> SOM compiler option.
```

Table 18-4. *C language specification comment styles.*

The first three styles you are probably familiar with. The third style, you recall, requires the use of a special compiler flag since it is not ANSI C. The fourth style is an OIDL comment, not a C comment. OIDL comments are not put into emitted files, so they do not have to be recognizable by a language compiler.

You may use the different styles interchangeably. However, you must be careful about where in the CSC file you place the comments. The SOM compiler places comments into emitted files based on where they occur in the CSC file, since OIDL syntax tells the SOM compiler which comments go with which statements. For example, if you want a method description to go with that method, you must put it after the method declaration. If you don't place them carefully according to OIDL syntax, they probably won't show up where you expect them to.

THE OIDL LANGUAGE—BASIC CSC SYNTAX

To define a class for use with SOM, you must write a CSC file in OIDL. This does not in any way detract from SOM's being language independent—you must define the class in OIDL, but you can implement the class in the language of your choice (as long as there are bindings for it). To define a class you must specify several things:

❑ The class name

❑ The metaclass name, if any

❑ The data, both public and private

- ❑ The methods, both public and private
- ❑ The class's parent

Since SOM generates files for you, you may also want to specify any of the following:

- ❑ Root for the file names generated
- ❑ Prefix for the method names
- ❑ Prefix for the class method names, if any
- ❑ Data to be put in any of the files SOM generates, including the header files

For backward compatibility, you may also specify what SOM calls a *release order*. This allows you to make modifications such as adding methods without having to recompile any existing clients.

Table 18-5 shows the sections contained in an OIDL file, in the recommended order for specifying them.

	REQUIRED	OPTIONAL
Include section	x	
Class section	x	
Metaclass section		x
Parent Class section	x	
Release Order section		x
Passthru section		x
Data section		x
Methods section		x

Table 18-5. *The sections of an OIDL file.*

The *Include* section is for including SC and PSC files. The *Class* section lets you define the name of the class and various options concerning it. The *Metaclass* section lets you specify the class for your class's class object. (The class object for your class will be an instance of the metaclass.) The metaclass will default to the metaclass of your class's parent. The *Parent Class* section lets you define what class you are subclassing, that is, who your class is directly descended from. SOM 1.0 does not support multiple inheritance, so you may specify only one parent. The *Release Order* section is to support backward compatibility. It is painless to use, and you will find it handy when you want to modify your class. The *Passthru* section lets you specify code to be put into the header files. Any data you would normally place in the header file by hand, such as a typedef, #define, or include statement, must be specified in the Passthru section. The *Data* and *Method* sections let you declare instance variables and methods.

Warning: Including the word *passthru* in a comment may confuse the SOM compiler. If you want to use the word, spell it out, as in *pass through*.

The complete syntax and description of OIDL, including these sections, are defined in detail later in this chapter in *Detailed CSC Reference*.

AN OIDL CLASS SPECIFICATION

Figure 18-2 contains a listing of the OIDL class specification for the calc class used in the previous chapter. It illustrates most of the basic parts of an OIDL class specification. Referring to Figure 18-2:

❑ Line 1 contains the INCLUDE section. Here you must put an include statement for the SC file of the parent. In this case the parent is the SOMObject class provided by the SOM runtime. Notice that there can be a difference between the name of the class itself and the filenames containing information about it.

❑ Line 3 contains the CLASS section. Here you may choose among several options but must at least specify the name of the class being defined. In this example, the name of the class is *calculator*.

❑ Lines 5 and 6 contain throwaway comments. These comments will not be placed into any of the files generated by the SOM compiler.

❑ Lines 7 and 8 are comments about the class itself. These comments will appear in the language-independent SC file generated by the SOM compiler. The SC file generated from this example is shown in Figure 18-3.

❑ Line 10 contains the PARENT section. In this example, the parent is the SOMObject class provided by the SOM runtime. The PARENT section and INCLUDE are closely related.

❑ Line 11 contains a comment about the parent. This comment will appear in the language-independent SC file generated by the SOM compiler. The SC file generated from this example is shown in Figure 18-3.

❑ Lines 13 and 14 are the RELEASE ORDER section. This section is optional. If you use it, you have the ability to add methods and data without having to recompile any programs that use your class. In your original CSC file, you should list all the methods, procedures, and non-internal data in your class. As you add methods or data to the CSC file, you should add them to the end of the release order section as well.

  Changing the order of items in the Release Order section will cause errors in existing applications if they are not recompiled. The version options in the class section can be used to indicate incompatibility between versions. (See *Detailed CSC Reference* later in this chapter.)

❑ Lines 16 through 23 are the PASSTHRU section. Since you should let the SOM compiler generate the header files, you should not edit them by hand to include such things as structure definitions or binary flags. Instead you would put those definitions in the passthru section, and the SOM compiler will automatically place them in the header file for you.

  Lines 16 through 18 show how you include other header files for your class implementation. The statement "#include <stdio.h>" will be put into the emitted IH file before the SOM-generated "#include <calc.ih>" statement. You could put these into the C file instead, but it is a good idea to put them into the header file. This keeps a consistent interface.

  Lines 20 through 23 show how you specify statements to be emitted into the H file. The H file will be included by anyone that uses your class. In this example, two error codes are defined. If you put them in the H file, they are accessible to both the class implementation and the client program.

❑ Lines 26 through 32 are the DATA section. Bindings for the data variables are put into the header files. If the data is declared to be internal, the binding goes into the IH file; if the data is declared to be private, the binding goes into the PH file; if the data is declared to be public, the binding goes into the H file.

Line 26 contains a comment about the data for this class. It will appear in the language-independent SC file generated by the SOM compiler if there is any public data and in the PSC file if there is any private data. The SC file generated from this example is shown in Figure 18-3, and the PSC file generated from this example is shown in Figure 18-4.

Line 27 contains the data statement. This tells the SOM compiler that data declarations follow.

Line 29 defines a private variable named Register1. The syntax for the declaration is ANSI C. This definition will be put into the PSC file.

Line 30 is a comment about the data variable defined just above, in line 29. It will appear in the PSC file just under the data declaration.

Lines 31 and 32 are similar to lines 29 and 30.

❑ Lines 35 through 54 are the METHODS section. Bindings for the methods are put into the header files. If the method is declared to be private, the binding goes into the PH file; otherwise the binding goes into the H file. It is very similar to the DATA section. Line 29 is the method statement itself. This example includes a throwaway comment and two different styles of comments which result in their being placed into the emitted files.

```
/**********************************************************************/
/*  prog18f2.c        calc.c   an oidl class specification         */
/**********************************************************************/

Line  1    #include <somobj.sc>
Line  2
Line  3    class: calculator;
Line  4
Line  5    ## This is where to put comments that describe
```

```
## the class
-- This class is for a calculator that can add
-- and subtract

parent: SOMObject;
-- This is a comment about the parent section

release order: Add, Subtract, Register1,
               Register2;

passthru:  C.ih, before;
   #include <stdio.h>
endpassthru;

passthru:  C.h, after;
   #define ERROR_OVERFLOW
   #define ERROR_UNDERFLOW
endpassthru;


-- Data
data:

     signed long    Register1,  private;
--                    A general-use register.
     signed long    Register2,  private;
--                    A general-use register.


methods:

-- Methods

signed long  Add(signed long var1,
                 signed long var2);

#  Put comments here that describe the Add method
-- Method Name:  Add
--   This method returns the sum of the two input
--   numbers.
--

signed long  Subtract(signed long var1,
                      signed long var2);

-- Method Name:  Subtract
//   This method returns the value of
//   var1 minus var2
//
```

Figure 18-2. *calc.csc (an OIDL class specification).*

```
# This file was generated by the SOM Compiler.
# FileName: calc.sc.
# Generated using:
#     SOM Precompiler spc: 1.22
#     SOM Emitter emitcsc: 1.10
#include <somobj.sc>

class: calculator;

    --  This class is for a calculator that can add
    --  and subtract


parent class: SOMObject;

    --  This is a comment about the parent section


release order:
    Add, Subtract, Register1,
    Register2;

passthru: C.h, after;
endpassthru;
methods:

    signed long    Add(signed long var1,
                signed long var2);

    --  Method Name:  Add
    --    This method returns the sum of the two input
    --    numbers.
    --

    signed long    Subtract(signed long var1,
                signed long var2);

    --  Method Name:  Subtract
    --    This method returns the value of
    --    var1 minus var2
```

Figure 18-3. *calc.sc (generated from the calc.csc in Figure 18-2).*

```
# This file was generated by the SOM Compiler.
# FileName: calc.psc.
# Generated using:
#     SOM Precompiler spc: 1.22
#     SOM Emitter emitpsc: 1.10

/*
 * Include the .sc file
 */
#include <calc.sc>
```

```
class: calculator;

data:
    signed long Register1, private;
    --                     A general-use register.

    signed long Register2, private;

    --                     A general-use register.
    --  Methods
```

Figure 18-4. *calc.psc (generated from the calc.csc in Figure 18-2).*

THE SOM-GENERATED C TEMPLATE

The SOM compiler generates a C file the first time you run it on a CSC file (if the emitter is selected in the SMEMIT environment variable). This file is a template for you to fill in with the code for implementing your class. If a C file already exists, the SOM compiler will add to it any new methods found in the CSC file but will not change the existing contents.

One important implication of this is that if you change the parameters of an existing method, you will have to make the adjustments in the C file yourself.

The C file generated by the SOM compiler contains three key elements: defines, includes, and method templates. It can be run through a C compiler as is without generating any errors. The C file resulting from calc.csc in Figure 18-2 is shown in Figure 18-5, with line numbers added for this discussion. Referring to Figure 18-5:

- ❑ Lines 2-8 are comments added by the SOM compiler. The style of comment can be set as a compiler option.

- ❑ Line 10 is a define added by the SOM compiler. You will find this define used in the header files that SOM generates. DO NOT erase it! If you suddenly start getting strange compile errors after extensively editing your C file, check to be sure this line is still intact.

- ❑ Line 11 is an include statement added by the SOM compiler. It causes the implementation header file, or IH file, to be included. The IH file contains an include statement for the H file. The IH file does not contain an include statement for the PH file. All of the information contained in the PH file is already in the IH, so it is not necessary to include both.

- ❑ Lines 13 through 18 are a comment describing the method. The comment is taken from the comment you put in the CSC file just below the method declaration. Notice that SOM changed the style of the comment, based on the compiler option.

- ❑ Lines 20 through 22 are the method declaration. Notice that there are three parameters, even though there are only two in the CSC file. The SOM compiler added the *calculator *somSelf* parameter in the first position. From your viewpoint as the class implementor, this parameter is a pointer to the object itself. From the viewpoint of a client using this class, the parameter is a pointer to the object on which the method is to be invoked.

  From here on, you treat it just as you would a regular C function, passing three arguments whenever you call it. A SOM convention is that the name of the first parameter to a method, the one representing the object itself, is always *somSelf*. You should not change this, since some of the SOM macro expansions refer to it by name.

- ❑ Lines 23 and 28 contain the braces for the beginning and end of the method. This is, of course, standard for a C file and enables you to delineate between methods.

- ❑ Line 24 declares and initializes a variable called somThis. Don't change its name, since the SOM macros use it. The variable is used to access all of the instance data for this object. If there is no instance data, then the SOM compiler will generate this line as a comment. If you later add instance data to the CSC file, you will need to uncomment it.

  Notice that the method name *calculatorGetData* is prefixed with the name of the class, as specified in the class statement of the CSC file. If you look at the header files, you will find the definition of the calculatorData structure. Neither your class nor a client should access it directly; always use somThis.

- ❑ Line 25 invokes a SOM-supplied debug routine. The SOM debug facilities are described in detail in the next chapter. Briefly, when you run the SOM trace facility, this statement will cause a line to be output including the class and method name. If you do not want the code that this adds, perhaps for performance reasons, you can set a define to turn it off.

❑ Line 27 is a return statement that returns 0. The SOM compiler always generates a return statement for you if the method is to return something. The value returned will be 0, cast to whatever type the method is supposed to return. Change this to the appropriate value.

```
Line  1
Line  2    /*********************************************************/
Line  3    /* This file was generated by the SOM Compiler.          */
Line  4    /* FileName: calc.c.                                     */
Line  5    /* Generated using:                                      */
Line  6    /*     SOM Precompiler spc: 1.22                         */
Line  7    /*     SOM Emitter emitc: 1.24                           */
Line  8    /*********************************************************/
Line  9
Line 10    #define calculator_Class_Source
Line 11    #include "calc.ih"
Line 12
Line 13    /*
Line 14     *  Method Name:  Add
Line 15     *    This method returns the sum of the two input
Line 16     *    numbers.
Line 17     *
Line 18     */
Line 19
Line 20    SOM_Scope signed long SOMLINK Add(calculator *somSelf,
Line 21              signed long var1,
Line 22              signed long var2)
Line 23    {
Line 24        calculatorData *somThis = calculatorGetData(somSelf);
Line 25        calculatorMethodDebug("calculator","Add");
Line 26
Line 27        return (signed long) 0;
Line 28    }
Line 29
Line 30    /*
Line 31     *  Method Name:  Subtract
Line 32     *    This method returns the value of
Line 33     *    var1 minus var2
Line 34     *
Line 35     */
Line 36
Line 37    SOM_Scope signed long SOMLINK Subtract(calculator *somSelf,
Line 38              signed long var1,
Line 39              signed long var2)
Line 40    {
Line 41        calculatorData *somThis =
                              calculatorGetData(somSelf);
```

```
        calculatorMethodDebug("calculator","Subtract");

        return (signed long) 0;
    }



```

Figure 18-5. calc.c (generated from the calc.csc in Figure 18-2).

C BINDINGS—USING A SOM CLASS

Once a class has been defined in SOM, programs can be written that use that class. In fact, another class's class implementation can use it! For now, just consider writing a normal program that uses a class.

SOM provides several macros and methods for client programs. The most common are the following:

```
<classname>NewClass(<x>, <y>)

<classname>New()

_<classname>

somNew(<class object>)

somFree(x)
```

The <classname>NewClass API is a function call that causes the class object associated with <classname> to be created. The SOM runtime will also be initialized if it has not been already. It takes two parameters which are a major version number and minor version number, respectively.

These version numbers are checked against those in the class's specification to be sure that the existing class version is one with which the client is compatible. If the version numbers are 0, any version is accepted. See the section later in this chapter, *Detailed CSC Reference,* for details of when versions are compatible.

No object instances can be created or worked with until the corresponding class object has been created. If a class object already exists when <classname>NewClass is called, it will simply return.

The <classname>NewClass function should never be called within another method call, such as _foo(barNewClass(0,0)). When method calls are expanded, the parameters may be evaluated more than once. The general rule is not to call the NewClass function in any expression that has side effects.

The <classname>New API is a macro that creates, or instantiates, an object of the <classname> class. The macro expands to call the <classname>NewClass API, so it is not necessary for you to do both. That is, when you use <classname>New, a class object will be automatically created if necessary.

The _<classname> macro is a shorthand way of referring to a class object. As you will see, there are many methods that can be invoked on a class object, including some that you as the class implementor can define. A client program must be sure that the class object exists before using this macro.

The somNew(<class object>) method is another way of instantiating an object. When the object it is invoked on is a regular class object, an instance of that class will be created. When the object it is invoked on is a metaclass object, a class object will be created, since instances of a metaclass are class objects. This method is useful when you don't know what type of object must be instantiated, for example, if it is a parameter to a function.

The somFree method is used to uninstantiate an object. When you no longer need the object in your client program, you invoke somFree on it. The somFree method will invoke the somUninit method. All memory and data pertaining to the object will be released.

Table 18-6 summarizes these macros and functions. Then Figure 18-6 and Figure 18-7 show two different ways a client program can create a calculator object.

MACRO/METHOD	PURPOSE
<classname>NewClass(<x>, <y>)	Create a class object.
<classname>New()	Instantiate an object.
_<classname>	Shorthand for the class object of a class.

MACRO/METHOD	PURPOSE
somNew(<class object>)	Instantiate an object.
somFreeObject	Destroy an object.

Table 18-6. *Client program APIs.*

```
#include <calc.h>
main (void)
{
   /* Declare "mycalc" to be a pointer to a calculator             */
   /* object.  Whenever you want to use a calculator               */
   /* object, use "mycalc", NOT "*mycalc"                          */

   calculator  *mycalc;

   mycalc = calculatorNew();    // Instantiates a calculator

}
```

Figure 18-6. *Instantiating a calculator object (one way).*

```
#include <calc.h>
main (void)
{
   /* Declare "mycalc" to be a pointer to a calculator             */
   /* object.  Whenever you want to use a calculator               */
   /* object, use "mycalc", NOT "*mycalc"                          */

   calculator  *mycalc;

   // Be sure the class object exists
   calculatorNewClass(0,0);

   // Invoke the somNew method on the class object
   mycalc = somNew(_calculator);
}
```

Figure 18-7. *Instantiating a calculator object (another way).*

C BINDINGS—ACCESSING INSTANCE DATA

SOM provides two ways of accessing instance data, one for the class implementor and one for the client. They allow you to use a shorthand and also provide encapsulation for the object. The mechanism is macros, defined in the header files. Within the class implementation, an instance variable is referenced by prefixing its name with an underscore. For example, the variable Register1 defined in calc.csc could be set as follows:

```
_Register1 = 10;
```

When run through a C compiler, this would be expanded to:

```
somThis->Register1 = 10;
```

Outside of the class implementation, a client would reference an instance variable by using a get macro. SOM provides a method for accessing an object's data outside of the class implementation. The get macro expands to invoke this method and can be used on either side of an assignment statement. The syntax of the get macro is:

```
get_<instance variable name>(<object>)
```

For example, to set the variable Register1 to two times the value in Register2:

```
get_Register1(calc1) = 2 * get_Register2(calc1);
```

where calc1 is a pointer to a calculator object.

The get macro is defined in either the public header file (H) or the private header file (PH), depending on whether the instance variable is defined to be public or private. This prohibits clients without access to the private header file from obtaining or altering the value of private data.

Figure 18-8 shows how you would fill in the Add method template for the calc class.

```
/**********************************************************************/
/*  Method Name:  Add                                                 */
/*    This method returns the sum of the two input numbers.           */
/**********************************************************************/

SOM_Scope signed long   SOMLINK Add(calculator *somSelf,
                  signed long var1,
                  signed long var2)
{
    calculatorData *somThis = calculatorGetData(somSelf);
    calculatorMethodDebug("calculator","Add");
    /* Start of non-SOM generated code                                    */
    _Register1 = var1;
    _Register2 = var2;
    /* Replace the template's return statement with                       */
    /* one of your own:                                                   */
    return (_Register1 + _Register2);
}
```

Figure 18-8. *calc.c fragment.*

Whenever a client (program) can directly access and change one of your object's instance variables there is a chance that the client will put an invalid value in it. If you want to avoid this possibility and the debugging nightmares it can bring, you should fully encapsulate all of the data for your class.

To do this, declare all of the data to be either private or internal and provide methods for users to retrieve or set the value. In this way, you can let your set method ensure that the value supplied is valid. This isn't only for SOM; full encapsulation is the suggested procedure for object-oriented programming.

If you want to be able to allow some users direct access to the data, declare it as private. Only those with access to the PH file will be able to directly change the data value. If you do not want anyone to have direct access to an instance variable, declare it as internal. No bindings are generated outside of the IH file for data declared as internal. Not even descendants have direct access to internal data.

One more word about instance data: You will usually want to initialize your instance data. The typical way to do this is to override the somInit and somUninit methods (do this in the methods section of the CSC file that declares the data). The somInit method is automatically invoked whenever an instance is created (by means of somNew or <classname>New), and somUninit is invoked when it is destroyed (by means of somFree).

C BINDINGS—INVOKING METHODS

SOM provides you with several ways to invoke a method on an object. The simplest way is to use the SOM macro of the form:

```
_<methodname>(object [,parameter1]*)
```

For example, if calc1 is a calculator object (an instantiation of the calculator class defined in calc.csc), you can invoke its Add method as shown in Figure 18-9.

```
...
#include <calc.h>

main (void)
{
   signed long sum;
   signed long num1 = 4;
   signed long num2 = 6;
   calculator  *calc1;

      /* Create a calculator object                                  */
   calc1 = calculatorNew();

      /* Invoke its Add method                                       */
   sum = _Add(calc1, num1, num2);                     /* sum is now 10 */

   somFree(calc1);
}
```

Figure 18-9. *One way to invoke a method.*

A second way to invoke a method is to use the SOM macro of the form:

```
<classname>_<methodname>(object [,parameter1]*)
```

For example, in the program above you could code:

```
sum = calculator_Add(calc1, num1, num2);
```

This form is useful when you want to be clear about what class the method belongs to. This form is necessary when you are using two different classes that each have the same method name, unless one is an override of the other or both methods were defined with the *name lookup* option. Otherwise the two method names will collide.

The C compiler will find two different macros for the method name, and you will get runtime errors when it is invoked. A section later in this chapter, *Tips,* explains how to deal with method name collisions.

If you do not know the name of the method in advance, you can still invoke it. To do this you must use either the somDispatch methods or the somID methods, described in the next chapter.

SUBCLASSING AND INHERITING IN SOM

Subclassing in SOM is very easy. In fact, you have already seen how to do this, since the *calculator* class is a subclass of SOMObject. (Recall that everything must be descended from SOMObject.) But, that's cheating. In this section you will see how to create a subclass of the calculator class.

Figure 18-12 is the class specification (CSC file) for the fcalc (fancycalculator) class from the previous chapter. Notice that the class name, fcalc, and the name of the file, fanccalc, do not have to be the same. The fcalc class is a subclass of calculator containing two new methods, multiply and divide.

Figure 18-16 shows the filled-in implementation for it. Notice that the statements defining somThis are commented out. The SOM compiler automatically commented them out because the fcalc class does not have any instance data of its own.

To illustrate one more point about subclassing, Figures 18-14 and 18-15 show the CSC and C files for dumbcalc, a subclass of fcalc. The hierarchy for the classes related to the calculator classes is shown in Figure 18-10 and Figure 18-11; dumbcalc overrides the multiply method of its parent, fcalc, to use the add method of calculator, its grandparent.

Notice that it invokes the add method on itself. It can do this because a dumbcalc is an fcalc is a calculator, so dumbcalc inherits the add method from its grandparent calculator.

Then Figure 18-16 shows how a client could use all three classes.

LEVEL	CLASS NAME	FILE
Root:	SOMObject	somobj
Child:	calc	calc
Grandchild:	fcalc	fanccalc
Greatgrandchild:	dumbcalc	dumbcalc

Figure 18-10. *The calculator class hierarchy.*

CLASS	PARENT	METHODS
calculator	SOMObject	Add,Subtract
fcalc	calculator	Add(I), Subtract(I), Multiply, Divide
dumbcalc	fcalc	Add(I),Subtract, Multiply(O), Divide(I)

KEY:

(I)	The method is inherited from an ancestor.
(O)	The method is inherited from an ancestor and overridden.

***Figure 18-11**. Methods in the calculator class hierarchy.*

```
#include <calc.sc>

class: fcalc;

parent: calculator;
-- This class is a subclass of the calculator class.
-- It inherits all of the calculator class's methods and data

release order: Multiply, Divide;

# Data
# This class does not contain any new data
data:

methods:

-- Methods

signed long  Multiply(signed long var1, signed long var2);

#  For simplicity, this method does not check for exceeding
#  maxint of the system
-- Method Name:  Multiply
--    This method returns the product of the two input numbers
--

double Divide(signed long var1, signed long var2);

-- Method Name:  Divide
--    This method returns the result of dividing the first
--    number by the second.  It will return 0 if the second
--    number is 0.
```

***Figure 18-12**. fanccalc.csc.*

```
/**********************************************************************/
/* This file was generated by the SOM Compiler.                       */
/* FileName: fanccalc.c.                                              */
/* Generated using:                                                   */
/*     SOM Precompiler spc: 1.22                                      */
/*     SOM Emitter emitc: 1.24                                        */
/**********************************************************************/

#define fcalc_Class_Source
#include "fanccalc.ih"

/*
 *  Method Name:  Multiply
 *   This method returns the product of the two input numbers
 *
 */

SOM_Scope signed long   SOMLINK Multiply(fcalc *somSelf,
                signed long var1,
                signed long var2)
{
    /* fcalcData *somThis = fcalcGetData(somSelf);                    */
    fcalcMethodDebug("fcalc","Multiply");

    return var1 * var2;
}

/*
 *  Method Name:  Divide
 *    This method returns the result of dividing the first
 *    number by the second.  It will return 0 if the second
 *    number is 0.
 *
 */

SOM_Scope double SOMLINK Divide(fcalc *somSelf,
          signed long var1,
          signed long var2)
{
    double res;

    /* fcalcData *somThis = fcalcGetData(somSelf);                    */
    fcalcMethodDebug("fcalc","Divide");

    if (var2 == 0) {
       res = (double) 0;
    } else {
       res = var1 / var2;
    }

    return res;
}
```

Figure 18-13. *fanccalc.c.*

```
#include <fcalc.sc>

class: dumbcalc;

parent: fcalc;

methods:

-- Methods

override Multiply;
```

Figure 18-14. *dumbcalc.csc.*

```
/*********************************************************************/
/* This file was generated by the SOM Compiler.                      */
/* FileName: dumbcalc.c.                                             */
/* Generated using:                                                  */
/*     SOM Precompiler spc: 1.22                                     */
/*     SOM Emitter emitc: 1.24                                       */
/*********************************************************************/

#define dumbcalc_Class_Source
#include "dumbcalc.ih"

/*
 *  Method Name:  Multiply
 *
 */

SOM_Scope signed long   SOMLINK Multiply(dumbcalc *somSelf,
                signed long var1,
                signed long var2)
{
   signed long counter = 0;
   signed long sum = 0;

   /* dumbcalcData *somThis = dumbcalcGetData(somSelf);              */
   dumbcalcMethodDebug("dumbcalc","Multiply");

   while (counter < var1) {
      sum = _Add(somSelf, sum, var2);
      counter++;
   }
   return sum;
}
```

Figure 18-15. *dumbcalc.c.*

```
#include <dumbcalc.h>

 main(argc, argv, envp)
    int argc;
    char *argv[];
    char *envp[];
 {
    signed long sum;
    signed long num1 = 4;
    signed long num2 = 6;
    calculator  *calc1;
    fcalc       *calc2;
    dumbcalc    *calc3;

       /* Create the calculator objects                              */

    calc1 = calculatorNew();
    calc2 = fcalcNew();
    calc3 = dumbcalcNew();

       /* The following are all valid method invocations             */
    sum = _Add(calc1, num1, num2);
    sum = calculator_Add(calc2, num1, num2);
    sum = calculator_Add(calc3, num1, num2);
    sum = _Multiply(calc2, num1, num2);
    sum = fcalc_Multiply(calc3, num1, num2);

 /* Destroy the calculator objects                                   */
    _somFree(calc1);
    _somFree(calc2);
    _somFree(calc3);
 }
```

Figure 18-16. *Using calc classes.*

METACLASSES

As a class implementor you will have times when you want to keep information about the class as a whole. For example, how many calculators are there? How many times has the Add method been invoked in this process?

Since a metaclass is a class in its own right, there are several ways to do this. The first thing you will need to do is specify a metaclass in your class's CSC file. Details about the metaclass can be put in either your class's CSC file or in a separate CSC file of its own. If the metaclass is defined in a separate CSC file, you will need to include its SC file in the INCLUDE section of your class's CSC.

A metaclass, being a class in its own right, can have instance data, methods, or both. Figure 18-17 shows how to define a metaclass within the same CSC file. It defines a metaclass for the *calculator* class called *metacalc*. The metacalc class will have one instance data variable and one method. Figure 18-18 shows the filled-in C file for the CSC in Figure 18-17, and Figure 18-19 shows a client program and sample output.

```
#include <somobj.sc>

class: calculator,
   function prefix = calcpre,
   class prefix =    metacalc;

-- The class prefix option is required when the metaclass
--    is included in the same CSC file.
-- The function prefix option is used here just to
--    illustrate it.

parent: SOMObject;
-- Notice that the parent and the metaclass are different!

release order: Add, Subtract, Register1,
               Register2, NumCalcs, NumAddCalls;
-- Notice that the overridden method, somNew, is not
-- listed in the release order!


         -- Data
data:

     signed long    Register1,  private;
--                    A general-use register.
     signed long    Register2,  private;
--                    A general-use register.
     short          NumCalcs, class, public;
--                    A running total.
     short          NumAddCalls, class, public;
--                    Also a running total.



-- Methods
methods:

signed long  Add(signed long var1,
                 signed long var2);

         #  Put comments here that describe the Add method
-- Method Name:  Add
--   This method returns the sum of the two input
--   numbers.
--
signed long  Subtract(signed long var1, signed long var2);
```

```
-- Method Name:  Subtract
//   This method returns the value of
//   var1 minus var2
//

override somNew, class;
-- Override it to increment NumCalcs

override somInit, class;
-- Override it to initialize data variables
```

Figure 18-17. *metacalc.csc (embedded metaclass in calc.csc).*

```
/*******************************************************************/
/* This file was generated by the SOM Compiler.                    */
/* FileName: metacalc.c.                                           */
/* Generated using:                                                */
/*     SOM Precompiler spc: 1.22                                   */
/*     SOM Emitter emitc: 1.24                                     */
/*******************************************************************/

#define calculator_Class_Source
#include "metacalc.ih"

/*
 *  Method Name:  Add
 *    This method returns the sum of the two input
 *    numbers.
 *
 */

SOM_Scope signed long   SOMLINK calcpreAdd(calculator *somSelf,
                signed long var1,
                signed long var2)
{
    calculatorData *somThis = calculatorGetData(somSelf);
    calculatorMethodDebug("calculator","calcpreAdd");

    /* increment the class instance data                        */
    /* Notice the use of the macro to reference the             */
    /*    class object                                          */
    _Register1 = get_NumAddCalls(_calculator);
    get_NumAddCalls(_calculator) = _Register1 + 1;

    /* Set the instance data                                    */
    _Register1 = var1;
    _Register2 = var2;

    return (_Register1 + _Register2);
}

/*
 *  Method Name:  Subtract
 *    This method returns the value of
 *    var1 minus var2
 *
 */

SOM_Scope signed long   SOMLINK calcpreSubtract(calculator *somSelf,
                signed long var1,
                signed long var2)
{
    calculatorData *somThis = calculatorGetData(somSelf);
    calculatorMethodDebug("calculator","calcpreSubtract");

    return var1 - var2;
}

#undef SOM_CurrentClass
#define SOM_CurrentClass SOMMeta

/*
 *  Override it to increment NumCalcs
 */

SOM_Scope SOMAny *  SOMLINK metacalcsomNew(M_calculator *somSelf)
{
    M_calculatorData *somThis = M_calculatorGetData(somSelf);
    M_calculatorMethodDebug("M_calculator","metacalcsomNew");

    _NumCalcs++;
    return (parent_somNew(somSelf));
}

/*
 *  Override it to initialize data variables
 */

SOM_Scope void   SOMLINK metacalcsomInit(M_calculator *somSelf)
{
    M_calculatorData *somThis = M_calculatorGetData(somSelf);
    M_calculatorMethodDebug("M_calculator","metacalcsomInit");

    /* Call the parent first!!!!                                     */
    parent_somInit(somSelf);

    /* Then initialize data variables                                */
    _NumCalcs = 0;
    _NumAddCalls = 0;
}
```

Figure 18-18. Implementation of metacalc.c

```
/********************************************************************/
/* This is the listing of client.c                                  */
/********************************************************************/

#inlcude <stdio.h>
#include <string.h>
#include "metacalc.h"

main(argc, argv, envp)
   int argc;
   char *argv[];
   char *envp[];
{
   signed long sum;
   calculator *mycalc;
   calculator *mycalc2;

   mycalc = calculatorNew();
   mycalc2 = calculatorNew();
   sum = _Add(mycalc, 5, 10);

   printf("The number of calculators is: %d\n",
            get_NumCalcs(_calculator));
   printf("The number of times add was called is: %d\n",
            get_NumAddCalls(_calculator));
}
   The output of client.exe is:

       The number of calculators is: 2
       The number of times Add was called is: 1
```

Figure 18-19. *Using metacalc.c.*

THE LIST EXAMPLE—PART 1

This list example corresponds to the list class hierarchy developed in Chapter 17. Part 1 consists of just the FIFOQ class. Figure 18-20 shows the CSC file for it, and Figure 18-21 shows the filled-in C file for it. Figure 18-22 shows a client program, and Figure 18-23 shows the client's output.

> **Note:** Two versions of the FIFIQ class are included on the diskette accompanying this book. To distinguish between them, the filename of the original version is FIFOQ_1.xxx. The filename of the new version is FIFOQ.xxx. The classname remains FIFIQ in both.

```
#include <somobj.sc>

class: FIFOQ;

parent:  SOMObject;

release order:  pListHead, pListTail,
    usListItemSize, AddItem, GetItem;

passthru: C.ih, before;
   #include <string.h>
   #include <os2.h>

   typedef struct ListItem {
                struct ListItem  *pNext;
                struct ListItem  *pPrev;
                UCHAR            cData;
           } ListItem, *pListItem;
endpassthru;

passthru: C.h, after;
   #define MEMORY_ERROR    1
   #define LIST_IS_EMPTY   2
   #define LIST_NOT_EMPTY  3
endpassthru;

data:
     pListItem    pListHead, private;
     pListItem    pListTail, private;
     USHORT       usListItemSize, public;
     BOOL         bListEmptyFlag, public;

methods:
     INT     AddItem(PVOID *pItem);
     --                  Adds an item to the list
     PVOID   GetItem();
     --                  Removes the current item from
     --                  the list and returns it

     override somInit;

     override somUninit;
```

Figure 18-20. *fifoq_1.csc (CSC file for the original FIFOQ class).*

```
/********************************************************************
/* This file was generated by the SOM Compiler.                     */
/* FileName: fifoq_1.c.                                             */
/* Generated using:                                                 */
/*     SOM Precompiler spc: 3.9                                     */
/*     SOM Emitter emitc: 2.3                                       */
/********************************************************************/
```

```
#define FIFOQ_Class_Source
#include "fifoq_1.ih"

/*
 *                    Adds an item to the list
 */

SOM_Scope INT   SOMLINK AddItem(FIFOQ *somSelf,
                PVOID *pItem)
{
    pListItem   pNewItem;
    INT         rc = 0;

    FIFOQData *somThis = FIFOQGetData(somSelf);
    FIFOQMethodDebug("FIFOQ","AddItem");

    do {
          /* Allocate space for a new ListItem                     */
       pNewItem = SOMMalloc(sizeof(ListItem)-sizeof(UCHAR) +
                         _usListItemSize);
       if (!pNewItem) {
          rc = MEMORY_ERROR;
          break;
       }

          /* Initialize the new Item                               */
       memcpy(&pNewItem->cData, pItem, _usListItemSize);
       pNewItem->pNext = (pListItem) NULL;
       pNewItem->pPrev = _pListTail;

          /* Add the new Item to the end of the List               */
       if (_bListEmptyFlag == LIST_IS_EMPTY) {
          _pListHead = _pListTail = pNewItem;
          _bListEmptyFlag = LIST_NOT_EMPTY;
       } else {
          _pListTail->pNext = pNewItem;
          _pListTail = pNewItem;
       }

    } while (0);        /* enddo                                   */

    return rc;
}

/*
 *                    Removes the current item from
 *                    the list and returns it
 */

SOM_Scope PVOID   SOMLINK GetItem(FIFOQ *somSelf)
{
    PVOID      retItem = NULL;
    pListItem  pSaveItem;
    FIFOQData *somThis = FIFOQGetData(somSelf);
    FIFOQMethodDebug("FIFOQ","GetItem");

    do {
       /* Error if the list is empty                                  */
       if (_bListEmptyFlag == LIST_IS_EMPTY) {
          break;
       }

       /* Copy the item from head of the chain                        */
       /* Allocate return buffer                                      */
       retItem = (PVOID) SOMMalloc( _usListItemSize);
       if (!retItem) {
          break;
       }
       memcpy(retItem, &_pListHead->cData, _usListItemSize);

       /* Remove the item from the list
        * Cases: 1 item in the list
        */       > 1 item in the list

       if (_pListHead == _pListTail) {
          /* Only one item in the list                                */
          _pListHead = _pListTail = (pListItem)  NULL;
          _bListEmptyFlag = LIST_IS_EMPTY;
       } else {
          _pListHead = _pListHead->pNext;
          _pListHead->pPrev = (pListItem) NULL;
       }

       /* Now free the list item itself                               */
       SOMFree(pSaveItem);

    } while (0 );          /* enddo                                   */
    return retItem;
}

SOM_Scope void   SOMLINK somInit(FIFOQ *somSelf)
{
    FIFOQData *somThis = FIFOQGetData(somSelf);
    FIFOQMethodDebug("FIFOQ","somInit");

    parent_somInit(somSelf);

    _pListHead = _pListTail = (pListItem) NULL;
    _usListItemSize = 1;
    _bListEmptyFlag = LIST_IS_EMPTY;

}

SOM_Scope void   SOMLINK somUninit(FIFOQ *somSelf)
{
    FIFOQData *somThis = FIFOQGetData(somSelf);
    FIFOQMethodDebug("FIFOQ","somUninit");
    while (_bListEmptyFlag != LIST_IS_EMPTY) {
       _GetItem(somSelf);
    }           /* endwhile                                          */

    parent_somUninit(somSelf);
}
```

Figure 18-21. *C file for the original FIFOQ class.*

```
#include <stdio.h>
#include <string.h>
#include <os2.h>
#include <fifoq_1.h>

main(argc, argv, envp)
   int argc;
   char *argv[];
   char *envp[];
{
   FIFOQ   *myq;
   USHORT  mydata[5] = {10, 15, 20, 25, 30};
   int      i;
   int     *j;
   int      rc;

   myq = FIFOQNew();
   SOM_Test(myq != NULL);

   get_usListItemSize(myq) = sizeof(USHORT);

   /* Now add and remove things from the list                       */
   for (i=0; i < 2; i++) {
      rc = _AddItem(myq, (PVOID) &mydata[i]);
      SOM_TestC(rc);
   }        /* endfor                                               */

   j = _GetItem(myq);
   printf("The first item was:  %d\n", *j);

   for (i=2; i < 4; i++) {
      rc = _AddItem(myq, (PVOID) &mydata[i]);
      SOM_TestC(rc);
   }        /* endfor                                               */

   /* Retrieve items                                                */
   for (i=0; i < 2; i++) {
      j = _GetItem(myq);
      printf("The next item was:  %d\n", *j);
   }        /* endfor                                               */

   rc = _AddItem(myq, (PVOID) &mydata[4]);

   for (i=0; i < 2; i++) {
      j = _GetItem(myq);
      printf("The next item was:  %d\n", *j);
   }          /* endfor                                             */

   _somFree(myq);
}
```

Figure 18-22. *client2.c (client program).*

```
 The first item was: 10
 The next item was:  15
 The next item was:  20
 The next item was:  25
 The next item was:  30
```

Figure 18-23. *Output of client program.*

THE LIST EXAMPLE—PART 2

This list example completes the list class hierarchy developed in Chapter 17. Items of interest are the following:

❑ Notice the release order statement in FIFOQ.CSC. Even though the methods and data were removed, they remain in this section of the CSC file.

  Note: Any clients compiled with the original FIFOQ class (from Part 1) will still work without having to be recompiled.

❑ Notice the AddItem overrides in FIFOQ and LIFOQ. They call the AddPos method which is only defined on their parent. However, they do not need to use the parent_AddPos macro.

Figures 18-24 and 18-25 show the CSC and C files for the LISTS class. Figures 18-26 and 18-27 show the CSC and C files for the revised FIFOQ class. Figures 18-28 and 18-29 show the CSC and C files for the LIFOQ class. Figures 18-30 and 18-31 show a client program that creates and uses a list of numbers as well as the output.

Note: On the diskette accompanying this book, the two versions of FIFOQ reside in DLLs with different names. For this reason, you would need to relink the client to use the new version.

```
#include <somobj.sc>

class: lists;

parent:  SOMObject;

release order:  pListHead, pListTail,
    usListItemSize, AddItem, GetItem, AddPos;

passthru: C.ih, before;
   #include <string.h>
   #include <os2.h>

   typedef struct ListItem {
                struct ListItem  *pNext;
                struct ListItem  *pPrev;
                UCHAR            cData;
           } ListItem, *pListItem;
endpassthru;

passthru: C.h, after;
   #define MEMORY_ERROR    1
   #define LIST_IS_EMPTY   2
   #define LIST_NOT_EMPTY  3
endpassthru;

data:
  pListItem    pListHead, private;
  pListItem    pListTail, private;
  USHORT       usListItemSize, public;
  BOOL         bListEmptyFlag, public;

methods:
     INT     AddItem(PVOID *pItem);
     --                 Adds an item to the list
     PVOID   GetItem();
     --                 Removes the current item from
     --                 the list and returns it

     INT     AddPos(PVOID *pItem, UCHAR pos);
     --                 Add to head or tail of list

     override somInit;

     override somUninit;
```

Figure 18-24. *lists.csc (CSC for the LISTS class).*

```
/******************************************************************/
/* This file was generated by the SOM Compiler.                  */
/* FileName: lists.c.                                             */
/* Generated using:                                               */
/*     SOM Precompiler spc: 1.22                                  */
/*     SOM Emitter emitc: 1.24                                    */
/******************************************************************/

#define lists_Class_Source
#include "lists.ih"

/*
 *                   Adds an item to the list
 */

/* All subclasses must override this method                       */
SOM_Scope INT   SOMLINK AddItem(lists *somSelf,
       PVOID *pItem)

    listsData *somThis = listsGetData(somSelf);
    listsMethodDebug("lists","AddItem");

    return (INT) 0;


/*
 *                    Removes the current item from
 *                    the list and returns it
 */

SOM_Scope PVOID   SOMLINK GetItem(lists *somSelf)
{
    PVOID      retItem = NULL;
    pListItem  pSaveItem;

    listsData *somThis = listsGetData(somSelf);
    listsMethodDebug("lists","GetItem");

    do {
       /* Error if the list is empty                               */
       if (_bListEmptyFlag == LIST_IS_EMPTY) {
          break;
       }

       /* Copy the item from head of the chain                     */
       /* Allocate return buffer                                   */
       retItem = (PVOID) SOMMalloc( _usListItemSize);
       if (!retItem) {
          break;
       }
       memcpy(retItem, &_pListHead->cData, _usListItemSize);

       /* Remove the item from the list                            */
       /* Cases: 1 item in the list                                */
```

```
        /*        > 1 item in the list                                  */

        if (_pListHead == _pListTail) {
           /* Only one item in the list */
           _pListHead = _pListTail = (pListItem)  NULL;
           _bListEmptyFlag = LIST_IS_EMPTY;
        } else {
           _pListHead = _pListHead->pNext;
           _pListHead->pPrev = (pListItem) NULL;
        }

        /* Now free the list item itself                                */
        SOMFree(pSaveItem);

     } while (0 );      /* enddo                                        */

     return retItem;
}

/*
 *                   Add to head or tail of list
 */

SOM_Scope INT   SOMLINK AddPos(lists *somSelf,
        PVOID *pItem,
        UCHAR pos)
{
    pListItem   pNewItem;
    INT         rc = 0;


    listsData *somThis = listsGetData(somSelf);
    listsMethodDebug("lists","AddPos");

    do {
           /* Allocate space for a new ListItem                         */
        pNewItem = SOMMalloc(sizeof(ListItem)-sizeof(UCHAR) +
                  _usListItemSize);
        if (!pNewItem) {
           rc = MEMORY_ERROR;
           break;
        }

           /* Initialize the new Item                                   */
        memcpy(&pNewItem->cData, pItem, _usListItemSize);
        pNewItem->pNext = (pListItem) NULL;
        pNewItem->pPrev = (pListItem) NULL;


        if (_bListEmptyFlag == LIST_IS_EMPTY) {
           _pListHead = _pListTail = pNewItem;
           _bListEmptyFlag = LIST_NOT_EMPTY;
           break;
        }
```

```
      if (pos == 'H') {
         /* Add the new Item to the head of the List                 */
         pNewItem->pNext = _pListHead;
         _pListHead = pNewItem;
      } else {
         /* Add the new Item to the end of the List                  */
         _pListTail->pNext = pNewItem;
         _pListTail = pNewItem;
      }               /* endif                                       */

   } while (0);       /* enddo                                       */

   return rc;
}

SOM_Scope void   SOMLINK somInit(lists *somSelf)
{
    listsData *somThis = listsGetData(somSelf);
    listsMethodDebug("lists","somInit");

    parent_somInit(somSelf);

    _pListHead = _pListTail = (pListItem) NULL;
    _usListItemSize = 1;
    _bListEmptyFlag = LIST_IS_EMPTY;


SOM_Scope void   SOMLINK somUninit(lists *somSelf)

    listsData *somThis = listsGetData(somSelf);
    listsMethodDebug("lists","somUninit");

    while (_bListEmptyFlag != LIST_IS_EMPTY) {
       _GetItem(somSelf);
    }                    /* endwhile                                 */

    parent_somUninit(somSelf);
}
```

Figure 18-25. *C for the LISTS class.*

```
#include <lists.sc>

class: FIFOQ;

parent:  lists;

release order:  pListHead, pListTail,
    usListItemSize, AddItem, GetItem;

passthru: C.ih, before;
   #include <os2.h>
endpassthru;
```

```
passthru: C.h, after;
endpassthru;

data:

methods:

override AddItem;
```

Figure 18-26. fifoq.csc (CSC for the new FIFOQ class).

```
  /*******************************************************************/
  /* This file was generated by the SOM Compiler.                    */
  /* FileName: fifoq.c.                                               */
  /* Generated using:                                                 */
  /*     SOM Precompiler spc: 3.9                                     */
  /*     SOM Emitter emitc: 2.3                                       */
  /*******************************************************************/

#define FIFOQ_Class_Source
#include "fifoq.ih"

  /*
   *                    Adds an item to the list
   */

SOM_Scope INT   SOMLINK AddItem(FIFOQ *somSelf,
        PVOID *pItem)
 {
     INT rc;

    /* FIFOQData *somThis = FIFOQGetData(somSelf);                   */
    FIFOQMethodDebug("FIFOQ","AddItem");

     rc = _AddPos(somSelf, pItem, 'T');
     return rc;
 }
```

Figure 18-27. C for the FIFOQ class.

```
#include <lists.sc>

class: LIFOQ;

parent:  lists;

release order:  AddItem;

passthru: C.ih, before;
   #include <os2.h>
endpassthru;
```

```
passthru: C.h, after;
endpassthru;

data:

methods:

override AddItem;
```

Figure 18-28. *LIFOQ.CSC: CSC for the LIFOQ class.*

```
/*********************************************************************/
/* This file was generated by the SOM Compiler.                      */
/* FileName: lifoq.c.                                                */
/* Generated using:                                                  */
/*     SOM Precompiler spc: 1.22                                     */
/*     SOM Emitter emitc: 1.24                                       */
/*********************************************************************/

 #define LIFOQ_Class_Source
 #include "lifoq.ih"

 SOM_Scope INT   SOMLINK AddItem(LIFOQ *somSelf,
        PVOID *pItem)
 {
 INT rc;

     /* LIFOQData *somThis = LIFOQGetData(somSelf);                  */
      LIFOQMethodDebug("LIFOQ","AddItem");

     rc = _AddPos(somSelf, pItem, 'H');
     return rc;
}
```

Figure 18-29. *C for the LIFOQ class.*

```
#include <stdio.h>
#include <string.h>
#include <os2.h>
#include <lifoq.h>

main(argc, argv, envp)
   int argc;
   char *argv[];
   char *envp[];
{
   LIFOQ   *myq;
   USHORT  mydata[5] = {10, 15, 20, 25, 30};
   int     i;
   int    *j;
   int     rc;
   listsNew();
   myq = LIFOQNew();
   SOM_Test(myq != NULL);

   get_usListItemSize(myq) = sizeof(USHORT);

   /* Now add and remove things from the list                                    */
   for (i=0; i < 2; i++) {
      rc = _AddItem(myq, (PVOID) &mydata[i]);
      SOM_TestC(rc);
   }                 /* endfor                                                   */

   j = _GetItem(myq);
   printf("The first item was:  %d\n", *j);

   for (i=2; i < 4; i++) {
      rc = _AddItem(myq, (PVOID) &mydata[i]);
      SOM_TestC(rc);
   }                 /* endfor                                                   */

   /* Retrieve items                                                             */
   for (i=0; i < 2; i++) {
      j = _GetItem(myq);
      printf("The next item was:  %d\n", *j);
   }                 /* endfor                                                   */

   rc = _AddItem(myq, (PVOID) &mydata[4]);

   for (i=0; i < 2; i++) {
      j = _GetItem(myq);
      printf("The next item was:  %d\n", *j);
   }                  /* endfor                                                  */

   _somFree(myq);
}
```

Figure 18-30. *client3.c (client program).*

```
The first item was: 15
The next item was:  25
The next item was:  20
The next item was:  30
The next item was:  10
```

Figure 18-31. *Output from the client3 program.*

```
#include <stdio.h>
#include <string.h>
#include <os2.h>
#include <lifoq.h>
#include <metacalc.h>
```

```
main(argc, argv, envp)
   int   argc;
   char  *argv[];
   char  *envp[];
{
   LIFOQ         *myq;
   calculator    *mycalc;
   signed long   sum;
   PVOID         tbuf;
   int           rc;

   listsNewClass(0,0);
   myq = LIFOQNew();
   SOM_Test(myq != NULL);

   get_usListItemSize(myq) = sizeof(PVOID);
   /* The list is now a list of pointers                                */

   /* Create two calculators and put them in the list                   */
   mycalc = calculatorNew();
   SOM_Test (mycalc != NULL);

   rc = AddItem(myq, (PVOID) &mycalc);
   SOM_TestC(rc);

   /*  That was one; now for the rest                                   */
   mycalc = calculatorNew();
   SOM_Test(mycalc != NULL);

   rc = _AddItem(myq, (PVOID) &mycalc);
   SOM_TestC(rc);

   /* Now use the calculators, first getting them off the list         */
   tbuf = GetItem(myq);
   SOM_Test(tbuf != NULL);
   mycalc = *((calculator **) tbuf);

   sum = _Add(mycalc, 5, 10);
   printf("The sum from the first calculator is:  %ld\n", sum);
   _somFree(mycalc);

   tbuf = GetItem(myq);
   SOM_Test(tbuf != NULL);
   mycalc = *((calculator **) tbuf);

   sum = _Add(mycalc, 1, 18);
   printf("The sum from the first calculator is:  %ld\n", sum);
   _somFree(mycalc);

   _somFree(myq);
}
```

Figure 18-32. *client4.c (client program using a list of calculators).*

DETAILED CSC REFERENCE

This section describes all of the options available in the OIDL. In most cases the order of optional parameters is unimportant, but in some cases this is not true. Therefore, even though you will be told when the order is required, it is recommended that you put the parameters in the order shown. If you use comments, it is always important to put them where the syntax says they can go. Except for comments beginning with a #, the SOM compiler places all comments in the OIDL file into some emitted file. Which emitted file, and where in the emitted file, is language dependent.

For consistency, the conventions used in this section to describe the syntax are the same as those used in both the online SOM documentation and the SOM reference manual in the OS/2 2.0 Technical Library. If you need more information about some topic, these are the best places to look. There are three distinct syntactic elements:

- ❑ Constants
- ❑ Metacharacters
- ❑ User supplied information

Constants are printed in **bold**. You should code constants in exactly the way they appear in the syntax definition (you don't have to make them boldface, though). Defaults are underlined.

Metacharacters are written in normal font. You should not code metacharacters. They are used to delineate options. There are 6 metacharacters, namely:

[ ] Delineates an optional part of the syntax
( ) Delineates a group of elements
| Delineates mutually exclusive elements
* Indicates that an element can be repeated 0 or more times

Examples of metacharacter usage follow:

(x | y)	Choose either x or y.
[x]	You can specify x or leave it out.

```
[ x | (y)* ]     You can specify either ONE x, or ANY number of
                 y's, or NOTHING.
```

The following are all valid choices: `x, y, yy, yyy`

User-supplied information is printed in *italics*. You should enter an expression that makes sense for your application. For example, if you are defining a class named "calculator" and saw the expression *classname* in the syntax in italics, you would specify the value *calculator*.

Include Section

This required section contains include statements for SC files, the language independent form of the public class specification. You must include the SC file for the class's parent and the SC file for the metaclass, if any. If you override a private method of any ancestor, you must include the ancestor's PSC file.

The order of the include statements is important, and is:

- ❑ First, the includes for ancestors, both SC and PSC. If there are more than one, the oldest ancestor (the one nearest the root) must come first.
- ❑ Second, the include for the parent
- ❑ Third, the include for the metaclass. (This is not needed here if the METACLASS section is included in the OIDL and it contains the *file stem* option. See the METACLASS section for more details.)

The syntax is the same as that in ANSI C:

```
[#include ( < ancestor >  | "ancestor")  ]*
#include  ( < parent >    | "parent")
[#include ( < metaclass > | "metaclass") ]*
```

Parameters:

- ❑ ancestor—The filename.ext of the SC or PSC file defining the method being overwritten.

- ❑ parent—The filename.ext of the SC or PSC file for the class's parent.
- ❑ metaclass—The filename.ext of the SC or PSC file for the class's metaclass.

If the filename is enclosed in quotation marks, the current directory will be searched first, followed by the directories in the SMINCLUDE environment variable. If the filename is enclosed in angle brackets, only the directories in the SMINCLUDE path will be searched.

Class Section

This required section contains the name of the class, several optional parameters for easing your life (at least, as a programmer), and optional parameters for backward compatibility. The order of the optional parameters is arbitrary, but if you include a description it must follow the *class* statement. The SOM compiler associates comments after the class statement as belonging to that class. It will place those comments after the class definition in the emitted SC file.

The syntax is:

```
class: name
   [, file stem = stem]
   [, external stem = stem]
   [, function prefix = prefix | external prefix = prefix]
   [, class prefix = prefix]
   [, major version = number]
   [, minor version = number]
   [, global | local]
   [, classinit = function];
[description]
```

Parameters:

- ❑ name—The name of the class being defined.
- ❑ file stem—The root filename for any files emitted by the SOM compiler (the SOM compiler determines which files to emit by the setting of the SMEMIT OS/2 environment variable). Also used as the class library name in a DEF file, if a DEF file is to be emitted. The default is the root filename of the OIDL file.

For example, compiling the OIDL file calc.csc with SMEMIT=c;h;ih would result in files named CALC.C, CALC.H, and CALC.IH.

- external stem—The root filename in the text for any files emitted by the SOM compiler (See filestem option); also allows for shortening of names. Since OS/2 2.1 allows names of up to 255 characters, this isn't usually necessary.

- function prefix—A prefix to be added to all method names in the class. The default is no prefix. For example, if function prefix is calc_ and the OIDL file contains a method named add, the full name of the method would be calc_add. You cannot specify both function prefix and external prefix. Methods defined on the metaclass, if any, are not included. Use the class prefix option for those.

- external prefix—A prefix to added to all method names in the class. Also causes method names to be external rather than local. The default is no prefix. You cannot specify both function prefix and external prefix. Methods defined on the metaclass, if any, are not included. Use the class prefix option for those.

- class prefix—A prefix to be added to the method name of any metaclass method. This is required if any metaclass methods are defined within the current class's OIDL file (in other words, if any method in the OIDL file's METHODS section has the "class" attribute). There is no default since this attribute is required if metaclass methods are specified and irrelevant if they are not.

- major version—One of two numbers checked at runtime to ensure code compatibility when a method is invoked. If the major version is 0, no check is done on it. If it is not 0, then the major version of the code being invoked must equal the major version of the code expected by the caller. (The version number is included in the H and IH files.) If used, the major version should be incremented whenever the new implementation is incompatible with existing clients. (For example, when a parameter is added to a method.) See also *minor version.*

  Note: You can specify a major version and not a minor version, or vice versa.

- ❑ minor version—One of two numbers checked at runtime to ensure code compatibility when a method is invoked. If the minor version is 0, no check is done on it. If it is not 0, then the minor version of the code being invoked must be equal to or greater than the minor version of the code expected by the caller. (The version number is included in the H and IH files.) If used, the minor version should be incremented whenever the new implementation is still compatible with existing clients. (For example, when a new method is added.) See also *major version*.

  Note: You can specify a major version and not a minor version, or vice versa.

- ❑ global—Emit code that does not look for other files in the local directory first. For example, the C file for the calc class would include the statement:

  ```
  #include <calc.ih>
  ```

  You cannot specify both global and local. Global is the default.

- ❑ local—Emit code that looks for other files in the local directory first. For example, the C file for the calc class would include the statement:

  ```
  #include "calc.ih"
  ```

  You cannot specify both global and local. Global is the default.

- ❑ classinit—A user-written function that is to be called when the class object is instantiated. (The somInit method of the class object will always be invoked.) If specified, a template for the function will be placed in the source code file. The default is none.

- ❑ description—A comment about the class. This comment will be placed in the emitted SC file (not in the emitted PSC file).

Metaclass Section

This optional section contains the name of the class's metaclass and any optional parameters for convenience and backward compatibility. You can also include a description, which must follow the *metaclass* statement. If a description is included, it will be placed after the metaclass statement in the emitted SC file. Do not include this section if the metaclass definition is not in its own OIDL file. This is the case when the metaclass is defined through the use of the *class* option on either a method (in the METHODS section) or data (in the DATA section). This section must be included if the metaclass definition is in its own OIDL file, even if it is modified here (through the use of the *class* option on either a method or data). The syntax follows:

```
metaclass: name
    [, file stem = stem]
    [, major version = number]
    [, minor version = number]
    [, global | local];
[description]
```

Parameters:

- ❑ name—The name of the metaclass for the class being defined.
- ❑ file stem—The root name of the files containing the metaclass. If this option is chosen, the #include metaclass.sc statement in the INCLUDE section is not necessary.
- ❑ major version—One of two numbers checked at runtime to ensure code compatibility when a method is invoked. If the major version is 0, no check is done on it. If it is not 0, then the major version of the code being invoked must equal the major version of the code expected by the caller. (The version number is included in the H and IH files.) If used, an error will be generated at runtime whenever the versions of this class and its metaclass are out of synch. See also *minor version*.
- ❑ minor version—One of two numbers checked at runtime to ensure code compatibility when a method is invoked. If the minor version is 0, no check is done on it. If it is not 0, then the minor version of the code being invoked must be equal to or greater than the minor version of the code expected by the caller. (The version number is included in the H and IH files.) If used, an error will be generated at runtime whenever this class is linked to an older version of the metaclass code. See also *major version*.

- global—Emit code that does not look for the metaclass files in the local directory first. For example, if the metaclass is in a file called mymeta.sc, then the emitted include statement would be the following:

  ```
  #include <mymeta.ih>
  ```

  You cannot specify both global and local. Global is the default.

- local—Emit code that looks for other files in the local directory first. For example, if the metaclass is in a file called MYMETA.SC, then the emitted include statement would be:

  ```
  #include "mymeta.ih"
  ```

  You cannot specify both global and local. Global is the default.

- description—A comment about the metaclass. This comment will be placed in the emitted SC file.

Parent Section

This required section contains the name of the class's parent and optional parameters for convenience and backward compatibility. You may also include a description, which must follow the *parent* statement. If a description is included, it will be placed after the parent statement in the emitted SC file.

The INCLUDE Section must contain an include statement for the parent's SC or PSC file. The syntax is:

```
parent: name
   [, file stem = stem]
   [, major version = number]
   [, minor version = number]
   [, global | local];
[description]
```

Parameters:

- name—The name of the parent for the class being defined.

- file stem—The root name of the files containing the parent.

- major version—One of two numbers checked at runtime to ensure code compatibility when a method is invoked. If the major version is 0, no check is done on it. If it is not 0, then the major version of the code being invoked must equal the major version of the code expected by the caller. (The version number is included in the H and IH files.) If used, an error will be generated at runtime whenever the versions of this class and its parent are out of synch. See also *minor version*.

- minor version—One of two numbers checked at runtime to ensure code compatibility when a method is invoked. If the minor version is 0, no check is done on it. If it is not 0, then the minor version of the code being invoked must be equal to or greater than the minor version of the code expected by the caller. (The version number is included in the H and IH files.)

  If used, an error will be generated at runtime whenever this class is linked to an older version of its parent's code. See also *major version*.

- global—Emit code that does not look for the parent's files in the local directory first. For example, if the parent is in a file called *myparent.sc*, then the emitted include statement would be:

  ```
  #include <myparent.ih>
  ```

  You cannot specify both global and local. Global is the default.

- local—Emit code that looks for files in the local directory first. For example, if the parent is in a file called myparent.sc, then the emitted include statement would be:

  ```
  #include "myparent.ih"
  ```

  You cannot specify both global and local. Global is the default.

- description—A comment about the parent. This comment will be placed in the emitted SC file.

Release Order Section

This optional section should contain a list of all of the public and private data, methods, and procedures defined in the file. Do not include methods that are overridden, since they are actually defined in the ancestor's OIDL. As new methods and data are added, they should be added to the end of this list.

This section defines the order in which items occur in externalized structures. The default is the order in which they occur in the OIDL file. Using this section allows new data and methods to be added to a class without affecting existing users of the class. If the release order is not used, the only way to maintain the compatibility is to add items only at the end of the OIDL file, which would prevent the logical grouping of methods and data. In general, not using this section means that all users of the class will have to recompile whenever new data or methods are added to the class.

> **Note 1:** One way to modify a class hierarchy is to move methods or data up to an ancestor. When this happens, the method or data should be added to the end of the Release Order statement for the ancestor but should remain in the original class's Release Order statement. (The only time an overridden method should be included in the Release Order statement is if it was left from such a modification.)
>
> **Note 2:** Do not include internal data members in the Release Order statement.
>
> **Note 3:** For the C language, the SOM compiler can emit a nicely formatted version of the OIDL file, complete with this section. To do so, include cs2 in the SMEMIT environment variable or in the -s option (the one that overrides the SMEMIT variable) when the SOM compiler is invoked.

The syntax is:

`release order:` [*name*], [*name*], . . . ;

Parameter:

- ❑ name—The name of a public or private method, procedure, or data member.

Passthru Section

This optional section contains information you want placed into emitted files before or after the SOM-generated #include statements (for example, this could be the #include CLASSNAME.IH statement in a .C file). Since the SOM compiler generates header files that you should not modify, you cannot place shared information in them directly.

For example, you often put typedefs and #defines in a header file. Use this section to put those typedefs, defines, and whatever into the various files.

The Passthru Section consists of one or more blocks of statements. The first line of each block specifies which file you want the statements to go in and whether you want them to go before or after any SOM-generated #include statements. You can include a before block and an after block for the same file.

All lines found within a block will be placed into the designated emitted file, regardless of what characters they contain. This means that comments, including throwaway comments, will show up in the emitted file.

For example, a common C language statement to put here is a #include. If SOM looked for comments, this statement would not end up in the emitted file, since the # symbol as the first in the line indicates to SOM that this is a throwaway comment.

> **Note 1:** In general, do not use the passthru statement to put lines into the C file. Instead, put them in the passthru for the IH file. Once the C file is generated, the SOM compiler modifies it only to add new methods, so changes to the passthru will be ignored.
>
> **Note 2:** Do not include the word *passthru* in a comment in the OIDL file. If you want to use it, spell the word out, for example, *pass through.* (It's a feature.)

The syntax for each block is:

```
passthru  language.extension, [before | after];
[line]*
endpassthru;
[description]
```

Parameters:

- ❑ language—The name of the programming language in which the class is to be implemented. The first version of SOM has bindings only for the C language.
- ❑ extention—The extension of the emitted file that will contain this block of statements. For the C language, the extension can be one of the following: C, H, PH, IH, SC, PSC, or CS2 (CS2 is the nicely formatted OIDL file). Passthru to the DEF file is not supported.
- ❑ before—Put these lines before any SOM-generated #include statements. This is the default. For example, you would probably want to include one of the C or OS2 header files before any others.
- ❑ after—Put these lines after any SOM-generated #include statements.
- ❑ line—A string of characters to be placed in the designated SOM-generated file.
- ❑ description—A comment about the passthru lines. This comment will be placed into the emitted file along with the passed-thru lines.

Data Section

This optional section lets you define any instance or class data you need. Each instantiation of your class will receive its own set of instance data. Class data, on the other hand, is unique to each instantiation of your class's metaclass. If you have data that is to be global to all instances, you should make it class data.

The syntax for the data section is:

```
data:
[description1]
[declaration  [,public | ,private | ,internal] [,class];
[description2]]*
```

Parameters:

- ❑ description1—A description encompassing all of the data being declared.

- ❑ declaration—For C, an ANSI C variable declaration.
- ❑ public—This variable is to be part of the public interface for the class. The binding for it will be placed in the H file. Anyone including the H file will have access to it.
- ❑ private—This variable is to be part of the private interface for the class. The binding for it will be placed in the PH file. Only those who include the PH file will have access to it.
- ❑ internal—This variable is not to be externalized outside of the class implementation file. The binding for it will be placed in the IH file. No one other than the class implementor will have access to it. This is the default.
- ❑ class—This variable is to be associated with the class object. The public, private, and internal options still apply.
- ❑ description2—A description about the variable declared above.

Methods Section

This optional section lets you define operations on your class. The operations can be functions or methods. As with data, methods can be public or private, and methods can be defined on either the class or the metaclass.

The syntax for the methods section follows:

```
methods:
[description1]
[[group: name;
   [description2]]
[method prototype
   [, public | , private ]
   [, method | , procedure ]
  [, class]
   [, offset  | , name lookup]
   [description3]]*
[override methodname
   [, public | , private ]
   [, class]
   [description4]]*
```

Parameters:

- ❏ description1—A description encompassing all of the methods being declared.
- ❏ name—The name for the following group of methods. If you have several methods, you may want to divide them into logical groups. For example, put error-handling routines in one group, memory-handling routines in another, and anything else in a third. The name is not used by SOM; it is to help you structure your design.
- ❏ method prototype—A prototype following the syntax of the particular language for the class implementation. For C, this is ANSI C syntax.
- ❏ description2—A description encompassing all of the methods in this particular group.
- ❏ public—This method is to be part of the public interface for the class. The binding for it will be placed in the H file. Anyone including the H file will be able to use it. This is the default.
- ❏ private—This method is to be part of the private interface for the class. The binding for it will be placed in the PH file. Only those who include the PH file will be able to use it.
- ❏ method—This is to be a method. When it is called, SOM will do method resolution processing. Subclasses can override it. This is the default.
- ❏ procedure—This is to be a function. When it is called, SOM will not do method resolution processing. Subclasses cannot override it. (Some programming languages make a distinction between functions and procedures based on whether or not they return anything. No such distinction is made here.)
- ❏ class—This method is to be associated with the class object. The public or private and method or procedure options still apply.
- ❏ description3—A description of the method declared above.

- ❑ override—The method named is defined in an ancestor class. Notice that you do not specify the prototype when the method is being overridden. SOM will determine the prototype from the ancestor class.
- ❑ public—Same as public described above.
- ❑ private—Same as private described above.
- ❑ class—Same as class described above.
- ❑ description4—A description of the method being overridden.

TIPS

- ❑ Declare data as private or internal and use encapsulation techniques whenever possible.
- ❑ Override somInit and somUninit whenever you have instance data.
- ❑ Override somInit and somUninit of the metaclass ehenever you have class instance data. You can do this simply by using the *class* option with the override, for example:

```
override SomInit, class;
```

- ❑ Override somNew whenever you have class instance data.
- ❑ Whenever you add a method, check the bottom of the C file. New method templates are always added to the bottom of the file. If you have class methods in the file, you must move any new instance methods above the following two lines:

```
#undef SOM_CurrentClass
#define SOM_CurrentClass SOMMeta
```

(An instance method is one that is not defined with the *class* option.)

❑ If you add a method to a class and get runtime errors when invoking it, compile with the warning flag on. You may have chosen a name that already exists in another class. If so, you can do one of three things:

First, you can choose another name.

Second, you can specify

```
#undef _<method name>
```

and forego the shorthand method invocation. Instead, invoke your method with the class name attached:

```
<classname>_<method name>
```

Third, you can modify the class definition file to add the *name lookup* option to the method definition. (This will impact performance slightly.)

❑ Whenever you change the parameters of an existing method, check the bottom of the C file. It may contain additional templates because the SOM compiler will not change existing code in the class implementation file. If you change the parameter in the CSC file and not in the C file, it will look to SOM like a new method was added. Accordingly, SOM will add a template for it at the bottom of the C file.

❑ If you use a debugger that does not recognize method names, be sure you are specifying the full name. The full name is the concatenation of the function prefix (from the OIDL CLASS section) and the method name. If the debugger still does not recognize the name, use the external prefix option in the OIDL CLASS section.

❑ To display instance data when using a debugger, display the value of somThis. somThis is a pointer to a structure containing all of the data defined on the class. If you are using the IPMD debugger, click on somThis to display its value as a pointer, and then click on it again to see the instance data values.

SUMMARY

This chapter introduces the System Object Model (SOM) and presents the advantages of SOM. The chapter discusses many of the basics of using SOM and showed the interaction between the development component and the runtime component.

The SOM compiler generates several types of files, each with a specific use. The SOM compiler also uses three environment variables that must be set. The syntax for running the SOM compiler and the syntax for the SOM specification language, Object Interface Definition Language (OIDL), are covered in detail.

Several examples illustrates how to build classes and class hierarchies. These examples shows how the SOM-generated files interact, how to apply object-oriented classes using SOM, and how to use metaclasses, subclasses, and inheritance. Sample programs also showed the methods, macros, and functions needed for implementing a class, using a class, and accessing the data of a class. Finally, techniques to assist in maintaining encapsulation are suggested.

CHAPTER 19

Selected SOM Topics

"SOM day my prince will come." —*Everywoman*

INTRODUCTION

Now that you've met SOM, it's time to get better acquainted. SOM has many features, including almost 100 methods, macros, and functions. You will not need many of these, but some will come in quite handy, and some are absolutely invaluable as you develop your classes. In this chapter, you will be shown approximately half of them.

The goal of this chapter is to cover specialized topics that you are likely to use, such as debugging and handling DLLs. To make this chapter useful as a reference, each section begins with a table showing which interfaces are covered in that section. The examples throughout this chapter are based on the class and client shown in Figures 19-1, 19-2, and 19-3.

SOM NAMING CONVENTIONS

SOM provides object-oriented support, along with a wealth of function and features, via a combination of macros, functions, and methods. There are also global variables that you may find useful. With a few exceptions, SOM uses the naming convention shown in Table 19-1.

Understanding the naming conventions will help you distinguish between somFree and SOMFree (the first is a method that frees the object on which it is invoked; the second is a function that frees memory at a specified location).

PREFIX	TYPE
som	method
SOM	function (pointer to a function)
SOM_	macro or global variable

Table 19-1. *SOM naming conventions.*

At this point it is worth taking a moment to review the difference between methods and functions. A method is a piece of code that is associated with a class and is invoked on an object of that class (or, through inheritance, a subclass of that class). A function is a piece of code that is not necessarily tied to any one class. It cannot be inherited or overridden.

SOM's mechanism for invoking a method on an object is to call a SOM function with that object as its first parameter. The SOM function in turn invokes the proper method on that object. If you look in the header files generated by the SOM compiler, you will see how this works. What you need to know is that macros and method calls are expanded, whereas function calls are not.

Debuggers will definitely know the difference. If you have problems with a debugger when using SOM, this hidden expansion may be part of the problem. For example, the IPMD debugger that comes with the IBM C Set2 ++ Version 2.0 compiler will display the value of a variable when you click on its name anywhere in the code. Since the name of an instance variable is actually a macro, the IPMD debugger will merely beep at you. (See the section later in this chapter, *Debugging SOM Programs,* for ways to get around this.)

A PROGRAMMING EXAMPLE

The code examples in this chapter define and use a class that implements strings. The string class has four methods:

❑ Set the value of the string.

❑ Clear the value of the string.

- ❑ Return the value of the string.
- ❑ Return the length of the string.

In this section, you will see the OIDL (SOM's Object Interface Definition Language) and the implementation for the string class, as well as how to call some of the most frequently used SOM methods. You will also see an example of a client program that uses the string class.

The OIDL file for this class is shown in Figure 19-1. Figure 19-2 shows the implementation for the string class after the programmer adds code to the method templates.

```
#
# STRNG.CSC -- OIDL file for the string class
#

#include <somobj.sc>

class:   String,
         file stem = strng;
parent:  SOMObject;

passthru: C.ih, before;
   #include <string.h>
endpassthru;

passthru: C.h, after;
   #define INVALID_PARAMETER  1
   #define MEMORY_ERROR       2
endpassthru;

data:
         int str_len;
         -- The string's length

         char * str;
         -- Pointer to the string

methods:
         int setString(char * ps);
         -- Set the string

         char * getString();
         -- Return the string value

         void clearString();
         -- Clear the string
         int getStringLen();
```

```
        -- Return the length of the string
        override somInit;
        -- Initialize the object

        override somUninit;
        -- Uninitialize the object

        override somDumpSelfInt;
        -- Dumps object instance data
```

Figure 19-1. *String.csc (OIDL for the string class).*

```
/********************************************************************/
/* This file was generated by the SOM Compiler.                    */
/* FileName: strng.c.                                               */
/* Generated using:                                                 */
/*     SOM Precompiler spc: 3.9                                     */
/*     SOM Emitter emitc: 2.3                                       */
/********************************************************************/

#define String_Class_Source
#include "strng.ih"

/*
 *  Set the string
 */

SOM_Scope int   SOMLINK setString(String *somSelf,
          char *ps)
{
    int rc = 0;

    StringData *somThis = StringGetData(somSelf);
    StringMethodDebug("String","setString");

    do {
       /* If a null pointer was passed, return error */
       /* Otherwise, clear any previous value,       */
       /* Allocate memory and set the instance data  */
       /* (Instance data are _str and _str_len )     */
       if (ps == NULL) {
          rc = INVALID_PARAMTER;
          break;
       }
       _clearString(somSelf);

       _str_len = strlen(ps);

       _str = SOMMalloc((_str_len)+1);
       if (_str == NULL) {
          rc = MEMORY_ERROR;
          break;
       }
```

```
        strcpy(_str, ps);
    } while(0);     /* Fall out of do loop */

    return rc;
}

/*
 *  Return the string value
 */

SOM_Scope char *  SOMLINK getString(String *somSelf)
{
    char * tmp = NULL;

    StringData *somThis = StringGetData(somSelf);
    StringMethodDebug("String","getString");

    do {
        if (_str == NULL) {
            break;
        }
        tmp = SOMMalloc((_str_len)+1);
        if (!tmp) {
            break;
        }
        strcpy(tmp, _str);

    } while(0);     /* Fall out of do loop */
    return tmp;
}

/*
 *  Clear the string
 */

SOM_Scope void  SOMLINK clearString(String *somSelf)
{
    StringData *somThis = StringGetData(somSelf);
    StringMethodDebug("String","clearString");

    // Free any allocated memory and reset the instance data
    SOMFree(_str);
    _str_len = 0;
    _str = NULL;
    return;
}

/*
 *  Return the length of the string
 */
SOM_Scope int   SOMLINK getStringLen(String *somSelf)
{
    StringData *somThis = StringGetData(somSelf);
    StringMethodDebug("String","getStringLen");
    /* The length is stored as instance data */
    return (int) _str_len;
}

/*
 *  Initialize the object
 */

SOM_Scope void   SOMLINK somInit(String *somSelf)
{
    StringData *somThis = StringGetData(somSelf);
    StringMethodDebug("String","somInit");

    parent_somInit(somSelf);     // Call the parent first

    _str_len = 0;                // Then initialize instance
    _str = NULL;                 // data
}

/*
 *  Uninitialize the object
 */

SOM_Scope void   SOMLINK somUninit(String *somSelf)
{
    StringData *somThis = StringGetData(somSelf);
    StringMethodDebug("String","somUninit");

    if (_str != NULL) {          // Free any memory allocated
       SOMFree(_str);            // for this object
    }
    parent_somUninit(somSelf);   // Call the parent last
}

/*
 *  Dumps object instance data
 */

SOM_Scope void   SOMLINK somDumpSelfInt(String *somSelf,
            int level)
{
    StringData *somThis = StringGetData(somSelf);
    StringMethodDebug("String","somDumpSelfInt");

    // Dump instance data after ancestors dump theirs
    parent_somDumpSelfInt(somSelf,level);

    somPrintf("\n str_len=%d", _str_len);
    somPrintf("\n str    =%s", _str);
}
```

Figure 19-2. *String.c (C implementation for the string class).*

If you are new to SOM and object-oriented programming, take some time to go over a few things about this example. The next paragraphs discuss:

- ❑ somInit and somUninit
- ❑ somDumpSelfInt
- ❑ parent_<methodname> calls and their locations
- ❑ The somSelf parameter
- ❑ The somThis macro
- ❑ Instance variables
- ❑ An encapsulation technique
- ❑ SOMMalloc and SOMFree

In the string class in Figure 19-1, somInit and somUninit are overridden. These methods, defined on the SOMObject class and inherited by all objects, are called when an object is created and freed, respectively. They are overridden in order to perform class-specific setup and clean up. You will frequently need to override these methods when defining your own classes. Note that the method somDumpSelfInt is frequently overridden also. This method is strictly for debugging. It will be discussed later in this chapter.

Notice the location of the parent calls in the three methods that are overridden. The parent_somInit call *must* be called as the first thing in the somInit method. The object is not fully ready for use until all of its parent classes have completed their initialization, including the SOMObject class. The parent_somUninit call must be called as the *last* thing in the somUninit method. After the parent classes have completed their uninitialization, you must not invoke any further methods on the object.

The parent_somDumpSelfInt call can be placed anywhere in the overridden method. Since the method displays data about the object for debugging purposes, you will want to be consistent about whether the data is displayed first from the parent and then from the child, or vice versa. The string example shows the data from the parent first, since it places the parent_somDumpSelfInt call prior to the other somPrintf calls.

Notice that the parameter *String *somSelf*, which was not in the method declarations in the OIDL file, was added to the parameter list of each method. This declares somSelf to be a pointer to an object of the *string* class. It is the object reference pointer that will be created when a string object is created by a client program. It is automatically added to the method templates by the SOM compiler.

When you invoke a method on a string object, you are actually calling a SOM function (look in the H file to find out how the macro expands into a function call). The first parameter of the SOM function is the object on which the method is to be invoked. The other parameters, if any, are the parameters to be passed to the method.

In object-oriented terms, you are sending a message to the object, by means of the SOM function, telling it to invoke a particular method. The fact that the object pointer appears within the class implementation code is due to the object model and the state of object-oriented C programming; in theory, the object knows who and where it is and therefore should not need a pointer to itself.

Where are the instance variables str_len and str? They are held in a structure in the object and are accessed by means of the somThis variable. somThis is declared and initialized by the call to StringGetData at the start of each method. This call is placed in the method template automatically by the SOM compiler. Never access the data in the structure directly!

As the class evolves, the structure may change, and your direct-access code will no longer be correct. Use your knowledge of the underlying structures only for debugging. The SOM compiler has generated the macros _str_len and _str, which are defined as (somThis->str_len) and (somThis->str). Youcan use these macros in your class implementation when referring to instance variables.

If a class does not contain any instance data when it is first defined, the somThis declaration and call to the <className>GetData function will be commented out in the emitted C file. If you do have instance data when you first generate the C file but later remove the instance data, you will have to comment out the calls yourself. The SOM 1.0 compiler will not change an existing C file in any way other than to add new method templates. If you remove all the instance data but leave the GetData calls uncommented, no harm will be done. However, if you start out with no instance data and later add some instance data, you must uncomment the GetData calls or else your compiler will complain that somThis is undefined.

Observe that in the getString method another buffer is allocated, the contents of the instance data _str are copied to it, and this new buffer is returned to the caller. Why not just return _str? Remember that one of the aims of object-oriented programming is encapsulation, that is, hiding instance data and preventing users from modifying it except through the methods provided. (Encapsulation, described in Chapter 18, also applies to structure definitions and methods.)

If you return _str, you are returning a pointer to the instance data itself, which would allow the user to change its value without calling setString or clearString. In fact, the user might even do so unknowingly, not realizing you had given it the object's data itself. To avoid this situation you should pass an intermediate buffer to protect the instance data.

Whenever you use this technique, you should document the fact that the client must free the buffer when it is through with it. If the _str data had been defined as private or internal, you would have no choice but to use this technique. As mentioned in Chapter 18, this is a good technique for keeping you and your code out of trouble.

> **Note:** If you have the problem of memory not being freed or being freed too often, you may be able to solve it by using an intermediate buffer.

Finally, notice the use of the function SOMMalloc to allocate the memory buffer for the string value in setString and getString. SOMMalloc is basically the same as the C malloc function. Note also that SOMFree is used to release the buffer in clearString and somUninit. SOMFree is basically the same as the C free function.

Now that you have a complete class implementation, you are ready to write a client program that tests and uses it. Figure 19-3 is a listing of a C program, Client1.c, which demonstrates the use of the string class.

```
/*****************************************************************/
/*  CLIENT1.C -- String class client program                     */
/*****************************************************************/

#include <stdlib.h>
#include <stdio.h>
```

```
#include "strng.h"

/*
 * Generic object state display routine.
 */
void dispString(char *s, int l)
{
   char *sp = "";

   if (s != NULL) {
      sp = s;
   }
   somPrintf("\nThe string is '%s'\n", sp);
   somPrintf("The string length is %d characters\n\n", l);
}

/*
 * MAINLINE
 */
main (long argc, char **argv)
{
   int     ln;
   char    *pstr;
   SOMAny *stringObj;

   StringNewClass(0, 0);         // Instantiate the string
                                 // class object, if it does
                                 // not already exist

   stringObj = StringNew();      // Instantiate a string
                                 // object

   _setString(stringObj,         // Set the value of the new
           "This is a string"); // object

   ln = _getStringLen(stringObj);  // Get the string's length
   pstr = _getString(stringObj);   // Get the string's value
   dispString(pstr, ln);           // Display the string's
                                   //    value
   SOMFree(pstr);                  // Free the intermediate
                                   //    buffer

   _clearString(stringObj);       // Clear the string
   ln = _getStringLen(stringObj); // Get the string's length
   pstr = _getString(stringObj);  // Get the string's value
   dispString(pstr, ln);          // Display the string's
                                  //    value

   _setString(stringObj,          // Set the value of the
     "Here is a new string");     //    object

   ln = _getStringLen(stringObj); // Get the string's length
   pstr = _getString(stringObj);  // Get the string's value
   dispString(pstr, ln);          // Display the string's
```

```
                                    //    value
   SOMFree(pstr);                   // Free the intermediate
                                     //    buffer

   _somFree(stringObj);             // Free the String object

   exit(0);

}
```

Figure 19-3. *Client1.c (client program for the string class).*

This client program's dispString function uses another SOM function, somPrintf. somPrintf is a SOM alternative function for C printf and has an interface identical to printf. The somPrintf output is channeled through SOMOutCharRoutine. All SOM functions, methods, and macros that output characters to the screen are channeled through SOMOutCharRoutine. The biggest advantage of using somPrintf is that SOMOutCharRoutine is a user-replaceable routine. These are all discussed later in this chapter, in *Debugging SOM Runtime*. (In case you are wondering, somPrintf is an exception to the naming convention.)

COMPILING AND LINKING A SOM CLASS

In this section you will see another example of compiling and linking class hierarchies. The string class will be built into a DLL.

Which emitters need to be run for this class? You always want the C, IH, and H files. Since this class may be subclassed, the SC file is needed. A DEF file is needed in order to link the DLL that will contain this class. Since the string class does not contain any private data or methods, you do not need a PSC or PH file.

In order for the SOM compiler to emit these files, you must either set the SMEMIT environment variable as follows:

```
set SMEMIT=def;ih;h;c;sc;
```

or you must execute the SOM compiler with the -s option like this:

```
sc -s def;ih;h;c;sc; strng.csc
```

In order to facilitate compiling and linking your programs, you probably use some kind of CASE tool. Make and nmake are two popular tools that run on DOS, OS/2 1.x, and OS/2 2.x.

Figure 19-4 is a make file for the string class and client programs in Figure 19-2 and Figure 19-3.

```
#
# MAK19_1.MAK
# Makefile for String class example 1
#
SC_OPTS=-sdef;ih;h;c;sc
CF_DLL=-c -Ti+ -Ge-
CF_EXE=-c -Ti+ -Ge+
LF=$(LF) /noi /pm:vio /debug /stack:8192
IMP_LIBS=som.lib
CLI_LIBS=som.lib+strng.lib

all         :  strng.dll client1.exe

strng.ih    :  strng.csc
               sc $(SC_OPTS) strng.csc

strng.obj   :  strng.c strng.ih
               icc $(CF_DLL) strng.c

strng.dll   :  strng.obj
               link386 $(LF) strng.obj,strng.dll,NUL,
                       $(IMP_LIBS),strng.def;
               implib strng.lib strng.def

client1.obj :  client1.c strng.ih
               icc $(CF_EXE) client1.c

client1.exe :  client1.obj
               link386 $(LF) client1.obj,client1.exe,NUL,
                        $(CLI_LIBS);
```

Figure 19-4. *mak19_1.mak (a make file for the string class and client1 program).*

This make file first compiles the OIDL file. Then it compiles the string class implementation program String.c, links it as String.dll, and creates an import library, String.lib. Then it compiles the client program Client1.c and links it as Client1.exe.

Note that both String.dll and Client1.exe are linked with Som.lib. Client1.exe is also linked with String.lib. Figure 19-5 shows the result of running Client1.exe.

```
The string is 'This is a string'
The string length is 16 characters


The string is ''
The string length is 0 characters


The string is 'Here is a new string'
The string length is 20 characters
```

Figure 19-5. *Output from client1.exe.*

DEBUGGING SOM PROGRAMS

SOM provides several facilities to help you in debugging your programs. Debuggers come in many different flavors, and you are bound to find one that fits your needs. This section will show you what the debugging aids are and how to use them.

Table 19-2 contains a summary of the facilities covered and their relationships to each other. You may find it useful to refer to when writing your code. Refer to Table 19-2 for the next few paragraphs. Figure 19-6 contains a client program, client2.c, that uses them. Client2 is the same as Client1 in Figure 19-3, with debug information added. You may find it useful to scan the examples there as you read this section.

NAME	PARAMETERS	TYPE	GV
SOM_TraceLevel		GV	
SOM_WarnLevel		GV	
SOM_AssertLevel		GV	
SOMOutCharRoutine	char	FU	
somPrintf	*	FU	
SOM_Error	e	MA	
SOM_WarnMsg	m	MA	GV2
SOM_Test	c	MA	
SOM_TestC	c	MA	GV2
SOM_Expect	c	MA	GV3
SOM_Assert	c,e	MA	GV3
somDumpSelf	*	ME	

NAME	PARAMETERS	TYPE	GV
somPrintSelf	*	ME	
somDumpSelfInt	*	ME	
<class>MethodDebug	*	MA	GV1

Key:

GV = Global Variable	GV1 = SOM_TraceLevel
FU = Function	GV2 = SOM_WarnLevel
MA = Macro	GV3 = SOM_AssertLevel
ME = Method	

Parameters:	c = condition
	e = error code
	m = message
	* = see method/macro definition

Table 19-2. Summary of debugging aids.

```
/***************************************************************/
/*  CLIENT2.C -- String class client program                   */
/*               using debugging information                   */
/***************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "strng.h"

extern SOM_TraceLevel;
extern SOM_WarnLevel;

/* All replacement SOM routines must use system linkage */
#pragma linkage(myOutCharRoutine, system)

/*
 * Replacement for SOMOutCharRoutine.
 * Writes SOM output to a log file.
 */
int myOutCharRoutine(char c)
{
   static FILE *log = NULL;

   if (log == NULL) {
      log = fopen("CLIENT2.OUT", "w");
   }
   putc(c, log);
   return(0);
}

/*
 * Generic object state display routine.
 */
void dispString(char *s, int l)
{
   char *sp = "";

   if (s != NULL) {
      sp = s;
   }
   somPrintf("\nThe string is '%s'\n", sp);
   somPrintf("The string length is %d characters\n\n", l);
}

/*
 * MAINLINE
 */
main (long argc, char **argv)
{
   int     dumpSelf = 0;
   int     ln;
   char   *pstr, *penv;
   SOMAny *stringObj;

   // Set the value of the SOM_TraceLevel global variable
   penv = getenv("SOM_TRACELEVEL");
   if (penv != NULL) {
      SOM_TraceLevel = atoi(penv);
   } else {
      SOM_TraceLevel = 0;
   }

   // Set the value of the SOM_WarnLevel global variable
   penv = getenv("SOM_WARNLEVEL");
   if (penv != NULL) {
      SOM_WarnLevel = atoi(penv);
   } else {
      SOM_WarnLevel = 0;
   }

   // If the SOM_LOG_OUTPUT environment variable is set,
   // then replace the SOMOutCharRoutine with one that sends
   // output to a file
   penv = getenv("SOM_LOG_OUTPUT");
   if (penv != NULL) {
      SOMOutCharRoutine = myOutCharRoutine;
   }

   // If the SOM_DUMP_SELF environment variable is set,
   // then set a flag
```

```
penv = getenv("SOM_DUMP_SELF");
if (penv != NULL) {
   dumpSelf = 1;
}

StringNewClass(0, 0);         // Instantiate the string
                              // class object, if it does
                              // not already exist

SOM_WarnMsg("Instantiating object");

stringObj = StringNew();      // Instantiate a string
                              // object
SOM_WarnMsg("Setting string");

_setString(stringObj,         // Set the value of the new
        "This is a string"); // object

ln = _getStringLen(stringObj);  // Get the string's length
pstr = _getString(stringObj);   // Get the string's value
dispString(pstr, ln);           // Display the string's
                                //    value
SOMFree(pstr);                  // Free the intermediate
                                //    buffer

SOM_WarnMsg("Clearing string");

_clearString(stringObj);       // Clear the string
ln = _getStringLen(stringObj); // Get the string's length

    // Two different ways to issue a generic warning
    // message if the string length is 0
SOM_TestC((ln > 0));
SOM_Assert((ln > 0), 1000+SOM_Warn);

pstr = _getString(stringObj);  // Get the string's value
dispString(pstr, ln);          // Display the string's
                               //    value

    // Yet another way, unless SOM_Error has been replaced
if (ln == 0) {
   SOM_Error(2000+SOM_Ignore);
   SOM_Error(3000+SOM_Warn);
   // SOM_Error(4000+SOM_Fatal); //
}

SOM_WarnMsg("Setting string");

_setString(stringObj,          // Set the value of the
  "Here is a new string");     //    object

ln = _getStringLen(stringObj); // Get the string's length
pstr = _getString(stringObj);  // Get the string's value
dispString(pstr, ln);          // Display the string's
```

```
                                   //    value
   SOMFree(pstr);                  // Free the intermediate
                                    //    buffer

   SOM_WarnMsg("Freeing object");

      // Conditional dump object information.  dumpSelf was
      // set above based on an environment variable
   if (dumpSelf) {
      _somDumpSelf(stringObj, 1);   // Dump all object info
                                    // include instance data
      somPrintf("\n");
      _somPrintSelf(stringObj);     // Only dump its class and
                                    // address
   }

   _somFree(stringObj);            // Free the String object
   exit(0);

}
```

Figure 19-6. *client2.c (a client program with debug information).*

SOM has two different kinds of debug facilities to help you. They are the *trace* facility and the *condition-testing* facility. SOM also lets you distinguish among three different levels of severity for errors. And SOM lets you turn the whole thing off at compile time, so you can eliminate any overhead in size or performance when you are ready to compile and link your application for your customers.

The Trace Facility for Debugging

There are times in debugging a program when you find it useful to know which methods have been called and in what order. You can generate such a trace by placing a print statement at the beginning of each method, which you must then remove before the final compilation of the program. SOM provides a more elegant solution.

The trace facility in SOM uses the <classname>MethodDebug macro along with the SOM_TraceLevel global variable. Set the SOM_TraceLevel global variable to certain nonzero values to activate a method trace. When you set SOM_TraceLevel to 1 or 2, a trace of the <className>MethodDebug method calls is produced.

When you set SOM_TraceLevel to a value of 1, only the calls in user-defined classes will be traced. When SOM_TraceLevel has a value of 2, the calls within the SOM runtime methods will be traced as well.

Using a value of 2 can give you some insight into how SOM works, if you are interested in the nuts and bolts. The format of a trace line's output is:

```
        "<file-name>":  <line-number>:  In <class:method>
```

where:

> <file-name> is the name of the file containing the <classname>MethodDebug call.
>
> <line-number> is the number of the line with the file
>
> <class:method> is the value taken from the two parameters on the call to <classname>MethodDebug.

This trace facility is particularly easy to use since the SOM compiler automatically puts a <classname>MethodDebug call at the beginning of each method template. In other words, all you have to do to produce a method trace is set the SOM_TraceLevel variable.

SOM_TraceLevel can be set and unset at various places in your program to control the trace output. The two typical ways to set it are by setting it directly to 0, 1, or 2 within your program (hardcoding it), or by setting it based on an OS/2 environment variable that you define.

You can change the value of SOM_TraceLevel as many times as you want. Since the variable is global to the process, you can set it within your class implementation code, your client program, or both. An example of setting the value of SOM_TraceLevel within a client program and based on a programmer-defined environment variable is shown in Figure 19-3.

Figure 19-7 shows the output of Client2.c when SOM_TraceLevel is set to 1. SOM_WarnLevel and SOM_AssertLevel, which are discussed later in this section, are both set to 0.

```
"STRNG.C": 97:   In String:somInit
"STRNG.C": 28:   In String:setString
"STRNG.C": 72:   In String:clearString
"STRNG.C": 88:   In String:getStringLen
"STRNG.C": 53:   In String:getString
```

```
The string is 'This is a string'
The string length is 16 characters
"STRNG.C": 72:   In String:clearString
"STRNG.C": 88:   In String:getStringLen
"STRNG.C": 53:   In String:getString

The string is ''
The string length is 0 characters
"STRNG.C": 28:   In String:setString
"STRNG.C": 72:   In String:clearString
"STRNG.C": 88:   In String:getStringLen
"STRNG.C": 53:   In String:getString

The string is 'Here is a new string'
The string length is 20 characters
"STRNG.C": 109:  In String:somUninit
```

Figure 19-7. *Output of Client2 (when SOM_TraceLevel = 1, SOM_WarnLevel = 0, SOM_AssertLevel = 0).*

The <classname>MethodDebug call does cause some overhead in the size and speed of your application, so you might want to compile it out when you are ready to compile the customer version of your application. Normally, you would do this by surrounding the call with the compiler directive "ifdef...endif". SOM lets you do it another way. To compile out the <classname>MethodDebug code, include the following statement before including the IH file:

```
    #define <classname>MethodDebug(c,m) SOM_NoTrace(c,m)
```

This will cause the <classname>MethodDebug macros to expand to nothing.

The Condition Testing Facility for Debugging

The condition testing facility consists of several macros which are summarized in Table 19-2. Just as the <classname>MethodDebug trace is dependent on the setting of the SOM_TraceLevel global variable, so some of the macros are also dependent on one of the global variables. These dependencies, when they exist, are noted in the third column of Table 19-2. For example, the SOM_WarnMsg macro will do nothing if the SOM_WarnLevel global variable is set to 0.

The macros let you test conditions and take appropriate action, print a warning message, or both. You can sprinkle these macros liberally throughout your code, compile them in, and then activate or deactivate them as you choose by setting the global variables they depend on at runtime. This may remind you of the technique of using printf statements for debugging. That's okay. The macros have much more power and flexibility, and you will find them much more useful than just a printf (although if you want to stay with just the SOM_Warn macro, that's almost the same).

The three different levels of severity for an error let you ignore it, put out a warning about it, or put out a message and terminate the process. SOM defines three constants, SOM_Ignore, SOM_Warn, and SOM_Fatal, corresponding to the three severity levels. Whenever you define an error code that you want to pass to SOM, you define the severity of the error code by setting its last digit to SOM_Ignore, SOM_Warn, or SOM_Fatal.

The rest of this section describes the six macros in detail.

SOM_Error

```
          SOM_Error(
                    int errcode);
```

Parameter:

- ❑ errcode (int) input—A numeric error code.

SOM_Error takes three pieces of information and passes them to some error handling routine. The three are: the error code that you define and provide, the source file name where SOM_Error was invoked, and the line number of the source file where it was invoked.

When you define an error code that will be used with SOM_Error the low-order digit of the error code should be set to SOM_Ignore, SOM_Warn, or SOM_Fatal. The default error-handling routine is SOMError, but you can replace SOMError if you wish. Replaceable functions are discussed in the next section, since there are many of them.

Client2.c in Figure 19-7 replaces SOMOutCharRoutine; you can replace SOMError in exactly the same way. The default SOMError formats the information passed to it and outputs it through the replaceable routine, SOMOutCharRoutine. The error code is then checked to see what the severity is. If it is a fatal error (the low order digit is SOM_Fatal) the process is terminated. Otherwise, control returns to the caller.

Following is a pseudo-code example showing two different ways to set the low-order digit. Figure 19-6 takes yet another shortcut for simplicity, but you will probably want to use one of the two methods below.

```
int errcode = 3000;

  /* 1: Setting the error severity based on a flag               */
if (IGNORE_ERROR) {
   errcode += SOM_Ignore;
} else if (WARN_ERROR) {
   errcode += SOM_Warn;
} else {
   errcode += SOM_Fatal;
}

SOM_Error(errcode);

/* 2: Setting the error severity at compile time                 */
#define ERRORBASE          3000
#define NITPROBLEM         (ERRORBASE + SOM_Ignore)
#define POSSIBLEPROBLEM    (ERRORBASE + SOM_Warn  )
#define BAAAADPROBLEM      (ERRORBASE + SOM_Fatal )
```

SOM_WarnMsg

```
          SOM_WarnMsg(
                      char *string)
```

Parameter:

- ❑ string (char *) input—Pointer to a null-terminated message string.

If SOM_WarnLevel has a value of 1 or 2, the input string is written by means of the replaceable routine SOMOutCharRoutine. Otherwise no action is taken. This is similar to a print statement encased in an "if . . . then . . . else" statement.

SOM_Test

```
          SOM_Test(
                   Boolean condition)
```

Parameter:

- ❑ condition (Boolean) input—A Boolean expression.

If the input condition is false (0), SOM_Error is invoked with a fatal error code, causing the process to terminate. Otherwise, execution continues.

SOM_TestC

```
SOM_TestC(
          Boolean condition)
```

Parameter:

- ❑ condition (Boolean) input—A Boolean expression.

If the input condition is false (0), two things happen. First, SOM_Error is invoked with a warning error code. Second, the value of SOM_Warnlevel is checked and if it is 1 or 2, a warning message is produced.

Notice that the difference between SOM_TestC and SOM_Test is twofold: First, SOM_TestC may or may not cause an additional message to be produced; and second, using SOM_TestC will never cause the process to terminate, even if the condition is false. This means you should use SOM_Test for very severe errors and SOM_TestC for lesser errors.

SOM_Expect

```
SOM_Expect(
           Boolean condition)
```

Parameter:

- ❑ condition (Boolean) input—A Boolean expression.

If the input condition is false (0), SOM_WarnMsg is called with a generic message. (Recall that SOM_WarnMsg outputs the message only in case the value of SOM_WarnLevel is 1 or 2.)

SOM_Assert

```
SOM_Assert(
           Boolean condition,
           int     errcode)
```

Parameters:

- ❑ condition (Boolean) input—A Boolean expression.
- ❑ errcode (int) input—A numeric error code.

If the input condition is false (zero), SOM_Error is invoked with the input error code. If it is a fatal error code the process will be terminated.

Comparing SOM_Assert with SOM_Test, SOM_TestC, and SOM_Expect, you will see that SOM_Assert lets you decide what the error code and its severity should be, whereas SOM_Test, SOM_TestC, and SOM_Expect do not.

The Client2.c program in Figure 19-6 contains calls to SOM_TestC, SOM_Assert, and SOM_WarnMsg. Figure 19-7 shows the output when SOM_TraceLevel is set to 1. Figure 19-8 shows the output when SOM_WarnLevel is set to 1, and Figure 19-9 shows the output when SOM_WarnLevel is set to 2. (To set that global variable for client2.c, you have to set the OS/2 global environment variable SOM_WARNLEVEL. SOM_WARNLEVEL is not part of the SOM debug facilities; it is defined and used only for this chapter.)

```
"CLIENT2.C": 36:  Warning: Instantiating object
"CLIENT2.C": 40:  Warning: Setting string

The string is 'This is a string'
The string length is 16 characters
"CLIENT2.C": 49:  Warning: Clearing string

The string is ''
The string length is 0 characters
"CLIENT2.C": 56: Warning, failed - (ln > 0).
"CLIENT2.C": 58: Warning: Assertion failed, code 0-100-1.
 ( (ln > 0) ).
"CLIENT2.C": 63: SOM Error - code = 0-300-1, severity =
Warning.
"CLIENT2.C": 69:  Warning: Setting string

The string is 'Here is a new string'
The string length is 20 characters
"CLIENT2.C": 76:  Warning: Freeing object
```

Figure 19-8. *Output of Client2 when (SOM_TraceLevel = 0, SOM_WarnLevel = 1, SOM_AssertLeve l= 0).*

```
"CLIENT2.C": 36:  Warning: Instantiating object
"CLIENT2.C": 40:  Warning: Setting string
The string is 'This is a string'
The string length is 16 characters
"CLIENT2.C": 49:  Warning: Clearing string

The string is ''
The string length is 0 characters
"CLIENT2.C": 56: Warning, failed - (ln > 0).
"CLIENT2.C": 58: Warning: Assertion failed, code 0-100-1.
 ( (ln > 0) ).
"CLIENT2.C": 62: SOM Error - code = 0-200-2, severity =
Information
only.
"CLIENT2.C": 63: SOM Error - code = 0-300-1, severity =
Warning.
"CLIENT2.C": 69:  Warning: Setting string

The string is 'Here is a new string'
The string length is 20 characters
"CLIENT2.C": 76:  Warning: Freeing object
```

Figure 19-9. *Output of client2 (when SOM_TraceLevel = 0, SOM_WarnLevel = 2, SOM_AssertLevel = 0).*

Just as you can compile out the <classname>MethodDebug calls, you can compile out these test macros. To do this, insert the following line before including the IH file:

```
        #define SOM_NoTest
```

Inherited Methods for Debugging

Table 19-3 shows the three methods that SOM provides explicitly for debugging.

somDumpSelf
somDumpSelfInt
somPrintSelf

Table 19-3. *SOM methods for debugging.*

These methods are defined on the SOMObject class and thus inherited by all other classes. They are used for printing information about an object, including its class, location, and instance data.

Accordingly, somDumpSelfInt is often overridden by any class that defines its own instance data while that class is still in the debugging stage. You may want to remove the override (in both the CSC and C files) prior to compiling the final, customer-ready version of the code to reduce the size of the method table.

When working with these methods, keep in mind that you are one link in a chain. That is, there is some hierarchy, starting with SOMObject and ending with either your class or a subclass of your class. Since every link in the chain gets a chance to print its data, you want to make sure that there is some order in which this occurs. This order explains why the parent should be called first. That way, the data is printed from the root, SOMObject, on down to the last subclass.

The rest of this section describes these three methods in detail.

somDumpSelf

```
void somDumpSelf(
                 SOMAny * object,
                 int    level)
```

Parameters:

- ❑ object (SOMAny *) input—A pointer to the object to be dumped.
- ❑ level (int) input—Print indentation level for information defined in this class.

somDumpSelf prints a line describing the object and then calls somDumpSelfInt. The description consists of the name of the object's class and the address of the object.

The print indentation level, which must be non-negative, provides a visual aid to distinguish among different levels in a class hierarchy by causing each level to be indented differently. Specifically, each line in the description will be indented by "2 X level" spaces.

somDumpSelfInt

```
void somDumpSelfInt(
                    SOMAny * object,
                    int    level)
```

Parameters:

- ❑ object (SOMAny *) input—A pointer to the object to be dumped.
- ❑ level (int) input—Print indentation level for information defined in this class.

somDumpSelfInt prints the object's instance data. For somDumpSelfInt to properly print the instance data, you must override it in any of your classes that add new instance data, be it internal, private, or public. In your override, you would first call the parent (parent_somDumpSelfInt) and then issue your own print statements for the instance data for your class. If you don't override somDumpSelfInt, the instance data for your class will not be printed; no other harm is done. If you look at String.c in Figure 19-2 you will see an example of overridding somDumpSelfInt.

The print indentation level, which must be nonnegative, provides a visual aid to distinguish among different levels in a class hierarchy by causing each level to be indented differently. Specifically, each line in the description will be indented by "2 X level" spaces.

somDumpSelfInt is not generally invoked directly, since printing instance data without printing which object the data belongs to isn't usually very helpful.

somPrintSelf

```
SOMAny * somPrintSelf(
                            SOMAny * object)
```

Parameter:

- ❑ object (SOMAny *) input—A pointer to the object to be printed.

somPrintSelf prints information similar to what somDumpSelf prints, but it does not call somDumpSelfInt. It also returns as a return value the pointer that was passed to it. If it is to your advantage to have a somDumpSelf type of method which returns the object pointer, there is no reason why you cannot override somPrintSelf and have your version invoke somDumpSelfInt. Figure 19-10 shows the output of Client2.c when the environment variable SOM_DUMP_SELF is set. (SOM_DUMP_SELF is not part of the SOM debug facilities; it is defined and used used only for this chapter.)

```
The string is 'This is a string'
The string length is 16 characters


The string is ''
The string length is 0 characters


The string is 'Here is a new string'
 The string length is 20 characters

1 {An instance of class String at address 00144F70

str_len=20
str    =Here is a new string 1 }
{An instance of class String at address 00144F70}
```

Figure 19-10. *Output of Client2 using somDumpSelf.*

REPLACEABLE FUNCTIONS

SOM provides the capability to replace certain routines. This means you can replace SOM-supplied functions with functions of your own choosing. This gives you more control over how your program performs certain tasks. Table 19-4 shows which routines are replaceable.

SOMCalloc
SOMClassInitFuncName
SOMDeleteModule
SOMError
SOMFree
SOMLoadModule
SOMMalloc
SOMOutCharRoutine
SOMRealloc

Table 19-4. *Replaceable routines.*

To replace a function, simply set the function address variable to the address of your user-written function.

An example follows.

```
            SOMCalloc = myCalloc;
```

SOMOutCharRoutine is at the core of all SOM character output and is called by routines such as somPrintf and SOMError. It takes one character as input and writes it. This is a function you will frequently want to replace, so that you can capture the output to a file or pipe it through another tool such as *more*. (You can always pipe the entire output of your program into a file, if you want the debugging output to be mixed in with the normal program output.)

The program in Figure 19-6 replaces SOMOutCharRoutine with a function called myOutCharRoutine. If the environment variable SOM_LOG_OUTPUT is set when this program is run, the program replaces SOMOutCharRoutine via the statement:

```
            SOMOutCharRoutine = myOutCharRoutine;
```

(SOM_LOG_OUTPUT is not part of the SOM debug facilities; it is defined and used only for this chapter.)

myOutCharRoutine will write output to the file CLIENT2.OUT instead of to stdout. The function myOutCharRoutine could just as easily be constructed to write the character both to the screen and to a file. SOM's flexibility is showing here. You can do anything you want to in your replacement function.

SOM IDs

SOM makes extensive use of what are called IDs. While you can write many applications without ever manipulating SOM IDs yourself, there are some applications that will require it. This section will give you a brief tour of SOM IDs so that if you ever do need to use them, you will know where to look.

If you look at the typedef for somId, you will find that it is a pointer. However, it is not always used as a pointer. SOM IDs are internal numbers that correspond to specially treated strings and are used by many SOM methods in lieu of the strings they represent. When SOM replaces a string with a SOM ID, it stores the string, as is, and returns an internally assigned number. Class names and method names are the most common types of strings that SOM must handle. (A SOM ID is also used to represent a descriptor, which is a string containing such information about a method as its number of parameters. These are used in dynamic dispatching and are required for writing a compiler or certain emitters. Descriptors are not covered in this book.)

Two SOM IDs are considered equal if the strings they represent are equal in a case-insensitive comparison. Table 19-5 summarizes some of the macros used to manipulate IDs.

`SOM_CheckId`
`SOM_CompareIds`
`SOM_StringFromId`
`SOMIdFromString`

Table 19-5. *SOM ID macros.*

If you look into how SOM works, you will find that, among other things, it maintains method tables for each class of object. You can invoke a method on an object in one of three ways:

- ❑ An offset into the method table of the object.
- ❑ A run-time comparison between a method name and the method names represented in the method table.
- ❑ A runtime dispatch routine.

The first two alternatives depend on which option you choose in the METHOD section of your CSC file, *offset* or name *lookup*. (The default is offset.) The third alternative is discussed later in this chapter in *Dispatching Methods*. It is also possible to define methods dynamically. All of this requires that SOM be able to uniquely identify methods as either numbers (offsets) or strings (names). This is where SOM IDs fit in.

There are two situations in which you may need to manipulate SOM IDs. First, you might want to invoke a method but not hardcode the methodname. For example, you might want to pass an object, method name, and parameters for the method to some general purpose routine and have that routine invoke the method. Second, you might want to create an object of a class that is not known until runtime, perhaps passed to you as a parameter.

The variable type *somId* is defined in SOM's header files which are automatically included when you subclass SOMObject. (As you know by now, all objects are subclassed off SOMObject). The value of a variable of this type can initially be a pointer to a string. After the ID is converted, it will contain an internal number and should no longer be used as a pointer. If you want to get the original string back, you can use one of the mcros described later in this section.

Figure 19-11, which follows the next section (*Class Hierarchy Methods*), contains an example of manipulating SOM IDs to create an object of an arbitrary class. The syntax for the SOM ID manipulation methods follows.

SOM_CheckId

```
        somId SOM_CheckId(
                              somId str)
```

Parameter:

- ❑ str (somId) input—A SOM ID that has not yet been converted into an internal form. (It is in the form of a pointer to a string.)

SOM_CheckId checks to see if the input string is already known. If it is, it returns the corresponding SOM ID for it. If it is not, it stores the string, creates a new SOM ID for it, and returns the new SOM ID. The function somRegisterID does the same thing and in addition returns 1 (true) if the ID was already known, 0 (false) if it was not.

SOM_CompareIds

```
        int SOM_CompareIds(
                              somId id1,
                              somId id2)
```

Parameters:

- ❑ id1 (somId) input—A SOM ID.

- ❑ id2 (somId) input—A SOM ID.

SOM_CompareIds returns 1 (true) if the two IDs represent the same string and 0 (false) if not. Since SOM ID comparisons are not case sensitive, comparing IDs created from the strings "STRING" and "string" returns true.

SOM_IdFromString

```
        somId SOM_IdFromString(
                               char * str)
```

Parameter:

- ❑ str (char *) input—A null-terminated character string.

SOM_IdFromString returns a SOM ID corresponding to the input string.

SOM_StringFromId

```
        char * SOM_StringFromId(
                                somId id)
```

Parameter:

- ❑ id (somId) input—A SOM ID.

SOM_StringFromId returns the string represented by the input ID.

CLASS HIERARCHY METHODS

SOM is largely a dynamic system. This means that many of the controlling data structures are generated at runtime rather than compile time. The advantage of this is that programs that use SOM can manipulate not only objects but classes themselves. This is possible because the class at runtime is embodied as an object, called a class object.

Another advantage is that the program can pass generic object pointers and dynamically determine specific information about the objects being pointed to, such as their class name and what classes they are descended from. Each SOM object, at runtime, has access to information about its class from its class's class object, as well as from itself.

SOM provides many methods and macros for accessing this information and for general object manipulation. The most common of these are summarized in Table 19-6.

somNew
somGetClass
somGetClassName
somIsA
somIsInstanceOf
somGetParent
somDescendedFrom
somFindClass

Table 19-6. *Summary of class hierarchy methods.*

The rest of this section describes the class hierarchy methods in detail.

somNew

```
SOMAny * somNew(
                    SOMAny * classObject)
```

Parameter:

- ❑ classObject (SOMAny *) input—A pointer to a class object.

somNew is invoked on a class object to create a new instance of that class. You would override this method if you wanted to maintain information about the instantiations of your class, such as how many of them existed. This is cleaner than having the newly instantiated object invoke a method on its class object (which you may need to do sometimes anyway). When you override somNew, you will almost always call the parent_somNew first. The calculator examples in Chapter 18 use somNew.

somGetClass

```
SOMAny * somGetClass(
                         SOMAny * object)
```

Parameter:

- ❑ object (SOMAny *) input—A pointer to the object.

somGetClass returns a pointer to the object's class object. You would need this when you wanted to invoke a method on the class object but did not know the name of the class beforehand. You will find this particularly useful when you have defined a metaclass and class methods. If you know the name of the class at compile time you do not have to use somGetClass to get a pointer to the class object. Instead, you can use the _<classname> macro to reference it.

somGetClassName

```
        char * somGetClassName(
                               SOMAny * object)
```

Parameter:

- ❑ object (SOMAny *) input—A pointer to the object.

somGetClassName returns a string containing the name of the object's class. There are several times when you might use this. You might want to ensure that an object was of a given class, you might want to know if two objects were of the same class, or you might merely want to display it. (If you just wanted to know if two objects were the same class but didn't want to use the class names for anything else, such as printing the names if they were not equal, you could use somGetClass and compare the pointers instead.)

somIsA

```
        int somIsA(
                   SOMAny * object,
                   SOMClass * classObject)
```

Parameters:

- ❑ object (SOMAny *) input—A pointer to the object.

- ❑ classObject (SOMClass *) input—A pointer to a class object.

somIsA returns a 1 (true) if an object is an instance of the class or is an instance of any of the class's descendants. It returns a 0 (false) if it is not. Notice the implicit link between an object and all of its ancestors. Because of inheritance, you can treat an object as though it is an instance of one of its ancestors. If you want to invoke a method on an object, it does not matter whether that object is of the first class to define the method or any of the subclasses of that first class.

somIsInstanceOf

```
          int somIsInstanceOf(
                                    SOMAny * object,
                                    SOMClass * classObject)
```

Parameters:

- ❑ object (SOMAny *) input—A pointer to the object.
- ❑ classObject (SOMClass *) input—A pointer to a class object.

somIsInstanceOf returns a 1 (true) if an object is an instance of the class and 0 (false) otherwise. The difference between somIsInstanceOf and somIsA is that if the object is a descendent of the specified class, somIsA will return true and somIsInstanceOf will return false. Depending on the information available at a certain time, you can choose among somIsA, somInstanceOf, and somGetClassName since they are closely related. To learn if an object is of a certain class when you know only a string representing that class, use somGetClassName. If you know the class object for the class, you can use somIsA or somInstanceOf.

somGetParent

```
          SOMClass * somGetParent(
                                    SOMAny * object)
```

Parameter:

- ❑ object (SOMAny *) input—A pointer to the object.

somGetParent returns a pointer to the class object for the object's parent class, with one exception. somGetParent returns only NULL if the input object is an instance of the

SOMObject class, since the SOMObject class is its own parent. Compare this method to somGetClass, which returns a pointer to the class object for the object itself.

somDescendedFrom

```
int somDescendedFrom(
                            SOMClass * classObject1,
                            SOMClass * classObject2)
```

Parameters:

- ❑ classObject1 (SOMClass *) input—A pointer to a class object.
- ❑ classObject2 (SOMClass *) input—A pointer to a class object.

somDescendedFrom returns a 1 (true) if the class generated by the first class object (through method calls such as somNew(classObject1)) is equal to or descended from the class generated by the second class object. Otherwise, somDescendedFrom returns 0 (false). This method is the same as somIsA except that it requires two class objects as input. Use whichever method is more convenient.

somFindClass

```
SOMClass * somFindClass(
                            SOMClassMgr * clsMgrObject,
                            somID classID,
                            int majorversion,
                            int minorversion)
```

Parameters:

- ❑ clsMgrObject (SOMClassMgr *) input—A pointer to an object of the SOMClass class or its descendants. While any class object for a metaclass is a descendant of the SOMClass class, you'll typically use somClassMgrObject, which is one of the global objects instantiated during SOM initialization. Client3.c in Figure 19-11 shows how to access it in C using the *extern* keyword.
- ❑ classID (somID) input—A SOM ID created from the class name to be found; must be the internal representation of the class name, not its representation as a pointer to a string.

- ❑ majorversion (int) input—The class implementation major version number or 0. If it is not 0, the class implementation that is located must have the same major version number. This is specified in the class's CSC file.

- ❑ minorversion (int) input—The class implementation minor version number or 0. If it is not 0, the class implementation that is located must have the same minor version number. This is specified in the class's CSC file.

somFindClass locates the class object for the specified class and returns a pointer to it. If the class object has not been created yet, somFindClass will cause it to be created. If the DLL that contains the class implementation has not been loaded yet, somFindClass will cause it to be loaded. See *DLLs* later in this chapter for more information.

Client3.c in Figure 19-11 and CLIENT4.C in Figure 19-13 show examples of how to use some of the class hierarchy methods discussed above. Client4.c uses the calculator classhierarchy defined in Chapter 18 to illustrate the use of these methods on subclasses. Figures 19-12 and 19-14 show the output of the two programs.

Client3.c uses these dynamic methods to create a new instance of a string object from an existing object and from a string class name. This will demonstrate that useful generic routines can be implemented using the SOM class hierarchy methods.

```
/*************************************************************/
/*  CLIENT3.C -- String class client program                 */
/*************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "strng.h"

extern SOMClassMgr *SOMClassMgrObject;

/*
 * Generic object state display routine.
 */
void dispString(SOMAny *obj)
{
   int     ln;
   char    *pstr, *s = "";

   ln = _getStringLen(obj);      // Get the string's length
   pstr = _getString(obj);       // Get the string's value
   if (pstr != NULL) {
       s = pstr;
   }
   somPrintf("\nThe string is '%s'\n", s);
   somPrintf("The string length is %d characters\n\n", ln);
   SOMFree(pstr);              // Free the intermediate buffer
}

/*
 *  Instantiate a new object from a existing one.
 */

SOMAny * objectFromObject(SOMAny * obj)
{
   SOMAny *newobj;
   SOMClass *classObj;

   classObj = _somGetClass(obj);    // Get the class object

   newobj   = _somNew(classObj);    // Create a new instance
                                    // of the class
   return(newobj);

}

/*
 *  Instantiate a new object from a class name.
 */

SOMAny * objectFromClassName(char * className)
{
   SOMAny *newobj;
   SOMClass *classObj;
   somId    classId;

   // Convert classname to Id
   classId  = SOM_IdFromString(className);

   //Find the class object
   classObj = _somFindClass(SOMClassMgrObject,
                            classId,
                            0, 0);

   // Create a new instance of the class
   newobj   = _somNew(classObj);
   return(newobj);

}

/*
 * MAINLINE
 */

main (long argc, char **argv)
{
   SOMAny *stringObj1, *stringObj2, *stringObj3;
```

```
    // Instantiate the class object
    StringNewClass(0, 0);

    // Instantiate a string object
    stringObj1 = StringNew();

    // Set the value of the object
    _setString(stringObj1, "Original object");

    // Display the object
    dispString(stringObj1);

    // Make a new object
    stringObj2 = objectFromObject(stringObj1);

      // Set the value of the object
    _setString(stringObj2,
                "This is an object from an object");

      // Display the new object
    dispString(stringObj2);

      // Display the original object
    dispString(stringObj1);

    // Make a new object
    stringObj3 = objectFromClassName("String");
     // Set the value of the object
    _setString(stringObj3,
                "This is an object from a class name");

    // Display the new object
    dispString(stringObj3);

    // Display the original object
    dispString(stringObj1);

    exit(0);
}
```

Figure 19-11. *Client3.c (using class hierarchy methods).*

```
/*************************************************************/
/*  CLIENT4.C -- Class Hierarchy Methods client program     */
/*************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "metacalc.h"
#include "fanccalc.h"
#include "dumbcalc.h"

extern SOMClassMgr *SOMClassMgrObject;
```

```
/* All replacement SOM routines must use system linkage      */
#pragma linkage(myOutCharRoutine, system)

/*
 * Replacement for SOMOutCharRoutine.
 * Writes SOM output to stdout
 */
int myOutCharRoutine(char c)
{

   printf("%c", c);
   return(0);
}

/*
 *  Display the name of an existing object's parent class
 */

void dispParentClass(SOMAny * obj)
{
// One way to get the name of an object's parent class
// using class hierarchy methods is:
// 1. Create an instance of the object's parent class
//    To do this, get the class object for the object's
//        parent class and invoke somNew on it.
// 2. Get the class name from the new instance
// 3. Destroy the instance created in step 1.
SOMClass *objClass;
SOMAny   *parentObj;
SOMClass *parentClassObj;
char     *parentClassName;
char     *objClassName;

objClass = _somGetClass(obj);
parentClassObj = _somGetParent(objClass);      // Step 1
parentObj =      _somNew(parentClassObj);

parentClassName = _somGetClassName(parentObj); // Step 2

objClassName = _somGetClassName(obj);
printf("The %s class's parent is the %s class\n",
         objClassName,  parentClassName);

_somFree(parentObj);                           // Step 3
}

/*
 * MAINLINE
 */

main (long argc, char **argv)
{
   calculator *calc1;    // An object of the calc class
   fcalc      *calc2;    // An object of the fcalc class
```

```
dumbcalc    *calc3;    // An object of the dumbcalc class
SOMAny      *anyObj;   // An object of any class

SOMOutCharRoutine = myOutCharRoutine;
// Instantiate three calculator class objects
calculatorNewClass(0, 0);
fcalcNewClass(0, 0);
dumbcalcNewClass(0, 0);

// Display the addresses of the SOM runtime class objects
printf("The four objects created by the SOM runtime
   are:\n");
printf("     1. The SOMObject class object: \n        ");
_somPrintSelf(_SOMObject);
printf("\n   2. The SOMClass class object: \n        ");
_somPrintSelf(_SOMClass);
printf("\n   3. The SOMClassMgr class object: \n        ");
_somPrintSelf(_SOMClassMgr);
printf("\n   4. The SOMClassMgr object: \n        ");
_somPrintSelf(SOMClassMgrObject);

// Display information about the calculator, fcalc,
// and dumbcalc class objects
printf("\n\n");
printf("The three calculator-related class objects
   are:\n\n");
printf("   5. For the \"calculator\" class:  \n        ");
_somPrintSelf(_calculator);
printf("\n   6. For the \"fcalc\" class:  \n        ");
_somPrintSelf(_fcalc);
printf("\n   7. For the \"dumbcalc\" class:  \n        ");
_somPrintSelf(_dumbcalc);

printf("\n\n\nThey are descended from the following
   classes:\n\n");
printf("   8. For the \"calculator\" class:  \n         ");
dispParentClass(_calculator);
printf("\n   9. For the \"fcalc\" class:     \n         ");
dispParentClass(_fcalc);
printf("\n   10. For the \"dumbcalc\" class:  \n         ");
dispParentClass(_dumbcalc);

// Now create three calculators, one for each class
calc1 = calculatorNew();
calc2 = fcalcNew();
calc3 = dumbcalcNew();

// Display information about them
printf("\n\nThe three calculator objects are:\n\n");
printf("calc1:    ");
_somPrintSelf(calc1);
printf("          ");
dispParentClass(calc1);
printf("\ncalc2:  ");
```

```
_somPrintSelf(calc2);
printf("          ");
dispParentClass(calc2);
printf("\ncalc3:  ");
_somPrintSelf(calc3);
printf("          ");
dispParentClass(calc3);

// Determine the hierarchy relationships between the
// three calculator objects, calc1, calc2, and calc3,
// and the three calculator classes, calculator,
// fanccalc, and dumbcalc
printf("\n\nResults of somIsA, somIsInstanceOf, and \
         somDescendedFrom calls:\n\n");
printf("calc1 and the calculator class:\n");
printf("     calc1  somIsA   \"calculator\" class?   %s\n",
          _somIsA(calc1, _calculator) ? "Yes" : "No");
printf("     calc1  somIsInstanceOf   \"calculator\"  class?
   %s\n", _somIsInstanceOf(calc1, _calculator) ? "Yes" :
   "No");
printf("     calc1  somDescendedFrom  \"calculator\"  class?
   %s\n\n", _somDescendedFrom(_calculator, _calculator) ?
   "Yes"     : "No");

printf("calc1 and the fcalc class:\n");
printf("     calc1  somIsA   \"fcalc\" class? %s\n",
             _somIsA(calc1, _fcalc) ? "Yes" : "No");
printf("     calc1  somIsInstanceOf   \"fcalc\" class?
      %s\n", _somIsInstanceOf(calc1, _fcalc) ? "Yes" :
      "No");
printf("     calc1  somDescendedFrom  \"fcalc\" class?
     %s\n\n", _somDescendedFrom(_calculator, _fcalc) ? "Yes"
     : "No");

printf("calc1 and the dumbcalc class:\n");
printf("     calc1  somIsA   \"dumbcalc\" class? %s\n",
             _somIsA(calc1, _dumbcalc) ? "Yes" : "No");
printf("     calc1  somIsInstanceOf   \"dumbcalc\" class?
      s\n", _somIsInstanceOf(calc1, _dumbcalc) ? "Yes" :
      "No");
printf("     calc1  somDescendedFrom  \"dumbcalc\" class?
     %s\n\n", _somDescendedFrom(_calculator, _dumbcalc) ?
     "Yes" : "No");

printf("calc2 and the calculator class:\n");
printf("     calc2  somIsA   \"calculator\" class? %s\n",
             _somIsA(calc2, _calculator) ? "Yes" : "No");
printf("     calc2  somIsInstanceOf   \"calculator\"class?
             %s\n", _somIsInstanceOf(calc2, _calculator) ?
             "Yes" : "No");

printf("     calc2  somDescendedFrom  \"calculator\"
   class? %s\n\n",  _somDescendedFrom(_fcalc, _calculator) ?
   "Yes" : "No");
```

```
printf("calc2 and the fcalc class:\n");
printf("     calc2  somIsA   \"fcalc\" class? %s\n",
             _somIsA(calc2, _fcalc) ? "Yes" : "No");
printf("     calc2  somIsInstanceOf   \"fcalc\" class?
      %s\n", _somIsInstanceOf(calc2, _fcalc) ? "Yes" :
      "No");

printf("     calc2  somDescendedFrom  \"fcalc\" class?
      %s\n\n", _somDescendedFrom(_fcalc, _fcalc) ? "Yes"
      :  "No");

printf("calc2 and the dumbcalc class:\n");
printf("     calc2  somIsA   \"dumbcalc\" class?
      %s\n", _somIsA(calc2, _dumbcalc) ? "Yes" : "No");
printf("     calc2  somIsInstanceOf   \"dumbcalc\" class?
      %s\n", _somIsInstanceOf(calc2, _dumbcalc) ?
      "Yes"  : "No");
printf("     calc2  somDescendedFrom  \"dumbcalc\" class?
       %s\n\n", _somDescendedFrom(_fcalc, _dumbcalc) ?
       "Yes" : "No");

printf("calc3 and the calculator class:\n");
printf("     calc3  somIsA   \"calculator\" class? %s\n",
             _somIsA(calc3, _calculator) ? "Yes" : "No");
printf("     calc3  somIsInstanceOf   \"calculator\"
        class? %s\n", _somIsInstanceOf(calc3, _calculator) ?
        "Yes" : "No");

printf("     calc3  somDescendedFrom  \"calculator\"
         class?  %s\n\n", _somDescendedFrom(_dumbcalc,
          _calculator) ? "Yes" : "No");

printf("calc3 and the fcalc class:\n");
printf("     calc3  somIsA   \"fcalc\" class? %s\n",
             _somIsA(calc3, _fcalc) ? "Yes" : "No");
printf("     calc3  somIsInstanceOf   \"fcalc\" class?
       %s\n", _somIsInstanceOf(calc3, _fcalc) ? "Yes" :
       "No");
printf("     calc3  somDescendedFrom  \"fcalc\" class?
       %s\n\n", _somDescendedFrom(_dumbcalc, _fcalc) ?
      "Yes" : "No");

printf("calc3 and the dumbcalc class:\n");
printf("     calc3  somIsA   \"dumbcalc\" class? %s\n",
             _somIsA(calc3, _dumbcalc) ? "Yes" : "No");
printf("     calc3  somIsInstanceOf   \"dumbcalc\" class?
     %s\n", _somIsInstanceOf(calc3, _dumbcalc) ?
    "Yes" : "No");

printf("     calc3  somDescendedFrom  \"dumbcalc\" class?
     %s\n\n", _somDescendedFrom(_dumbcalc, _dumbcalc) ?
     "Yes" : "No");

// Destroy the calculator objects
```

```
   _somFree(calc1);
   _somFree(calc2);
   _somFree(calc3);

   exit(0);
}
```

Figure 19-12. *Client4.c (using class hierarchy methods).*

```
The string is 'Original object'
The string length is 15 characters


The string is 'This is an object from an object'
The string length is 32 characters


The string is 'Original object'
The string length is 15 characters


The string is 'This is an object from a class name'
The string length is 35 characters


The string is 'Original object'
The string length is 15 characters
```

Figure 19-13. *Output of Figure 19-11 (Client3.c).*

The following items pertain to Figure 19-14.

```
The four objects created by the SOM runtime are:

     The SOMObject class object:
     An instance of class SOMClass at address 00220090}

     The SOMClass class object:
     {An instance of class SOMClass at address 00220110}

     The SOMClassMgr class object:
     An instance of class SOMClass at address 00224010}

     The SOMClassMgr object:
     {An instance of class SOMClassMgr at address 00224950}

The three calculator-related class objects are:

     For the calculator class:
     {An instance of class M_calculator at address 00224D90}
     For the fcalc class:
     {An instance of class M_calculator at address 00228090}

     For the dumbcalc class:
     {An instance of class M_calculator at address 00228410}

They are descended from the following classes:

     For the calculator class:
     The M_calculator class's parent is the SOMClass class.
     For the fcalc class:
     The M_calculator class's parent is the SOMClass class.

     For the dumbcalc class:
     The M_calculator class's parent is the SOMClass class.

The three calculator objects are:

calc1:   {An instance of class calculator at address 002286B0}
         The calculator class's parent is the SOMObject class.

calc2:   {An instance of class fcalc at address 002286D0}
         The fcalc class's parent is the calculator class.

calc3:   {An instance of class dumbcalc at address 002286F0}
         The dumbcalc class's parent is the fcalc class.

Results of somIsA, somIsInstanceOf, and somDescendedFrom calls:

calc1 and the calculator class:
       calc1  somIsA   "calculator" class?               Yes
       calc1  somIsInstanceOf   "calculator"  class?     Yes
       calc1  somDescendedFrom  "calculator"  class?     Yes

calc1 and the fcalc class:
       calc1  somIsA   "fcalc" class?                    No
       calc1  somIsInstanceOf   "fcalc" class?           No
       calc1  somDescendedFrom  "fcalc" class?           No

calc1 and the dumbcalc class:
       calc1  somIsA   "dumbcalc" class?                 No
       calc1  somIsInstanceOf   "dumbcalc" class?        No
       calc1  somDescendedFrom  "dumbcalc" class?        No

calc2 and the calculator class:
    calc2  somIsA   "calculator" class?                  Yes
    calc2  somIsInstanceOf   "calculator" class?         No
    calc2  somDescendedFrom  "calculator" class?         Yes

calc2 and the fcalc class:
    calc2  somIsA   "fcalc" class?                       Yes
    calc2  somIsInstanceOf   "fcalc" class?              Yes
    calc2  somDescendedFrom  "fcalc" class?              Yes
```

```
calc2 and the dumbcalc class:
    calc2  somIsA   "dumbcalc" class?                       No
    calc2  somIsInstanceOf   "dumbcalc" class?              No
    calc2  somDescendedFrom  "dumbcalc" class?              No

calc3 and the calculator class:
    calc3  somIsA   "calculator" class?                     Yes
    calc3  somIsInstanceOf   "calculator" class?            No
    calc3  somDescendedFrom  "calculator" class?            Yes

calc3 and the fcalc class:
    calc3  somIsA   "fcalc" class?                          Yes
    calc3  somIsInstanceOf   "fcalc" class?                 No
    calc3  somDescendedFrom  "fcalc" class?                  Yes

calc3 and the dumbcalc class:
    calc3  somIsA   "dumbcalc" class?                        Yes
    calc3  somIsInstanceOf   "dumbcalc" class?               Yes
    calc3  somDescendedFrom  "dumbcalc" class?               Yes
```

***Figure 19-14**. Output of Figure 19-12 (Client4.c).*

MEMORY MANAGEMENT FUNCTIONS

SOM provides four memory management functions, shown in Table 19-7. These are essentially ANSI C functions with SOM wrappers, but they have three big advantages:

- ❑ Additional error checking
- ❑ Language independence
- ❑ Replaceability

These are actually functions, not methods, so you don't pass an object pointer as the first parameter. The syntax for them is the same as that for the ANSI C functions.

SOMCalloc	Allocate buffer of size num x len
SOMFree	Free buffer
SOMMalloc	Allocate buffer of size
SOMRealloc	Reallocate buffer

***Table 19-7.** Memory management functions.*

Whenever an error occurs in one of these functions, SOMError is called. SOMError is described earlier in this chapter as a replaceable routine.

Language independence means that you can call these functions from any language that has SOM bindings, not just C. This is significant, since you may want to allocate memory in one class and free it somewhere else. However, if you allocate space one way and free it a different way, you may end up with all kinds of trouble. For example, when you allocate space in C, several variables and tables are initialized, set, or both. If you then free the space in Smalltalk, those C tables are not updated.

Furthermore, any SmallTalk tables won't know about the space since they weren't updated during the allocate. Since the SOM functions are language-independent, you are not restricted to C programs when calling them, so this kind of problem does not occur.

Replaceability means that all four functions can be replaced with functions of your choosing. You could decide to add even more error checking, or you could decide to use another language's allocate/deallocate scheme, or you could implement a pool/subpool memory scheme. With this flexibility, you can do whatever is appropriate for your application. (These functions are replaced in the same manner as the somOutCharRoutine method, which is demonstrated in Figure 19-6. Replaceable routines in general are discussed earlier in this chapter in *Replaceable Routines.*)

As a general rule, if you replace one of these functions you should replace all four of them. This is to prevent the kind of conflict that was avoided by SOM's being independent of a certain language; you don't want to reintroduce it! The syntax of the four Memory Management functions follows.

SOMCalloc

```
void * SOMCalloc(
                 size_t num,
                 size_t len)
```

Parameters:

- ❑ num (size_t) input—Number of elements.
- ❑ len (size_t) input—Length of each element in bytes.

SOMFree

```
void SOMFree(
             void * ptr)
```

Parameter:

- ❑ ptr (void *) input—Pointer to buffer to free.

SOMMalloc

```
void * SOMMalloc(
                 size_t size)
```

Parameter:

- ❑ size (size_t) input—Number of bytes to allocate.

SOMRealloc

```
void * SOMRealloc(
                 void * ptr,
                 size_t len)
```

Parameters:

- ❑ ptr (void *) input—Pointer to existing buffer.
- ❑ len (size_t) input—New length of buffer in bytes.

DISPATCHING METHODS

SOM provides a set of methods for dynamically determining on what object a method should be invoked and how the method should be invoked. These methods are summarized in Table 19-8. Each method supports a different type of return variable, the four types being void, void *, double, and long. These methods are useful for invoking methods when the names and number of arguments are not known at compile time. They are also useful for implementing user-defined dispatching algorithms.

This chapter assumes that the default SOM-supplied dispatching algorithms are used. Also, unless otherwise specified, the term *method* refers to the method to be invoked, not the somDispatchX method.

METHOD	RETURNED TYPE
somDispatchA	void *
somDispatchD	double
somDispathL	long
somDispatchV	void (no returned value)

Table 19-8. *Dispatching methods.*

As a group, these dispatching methods are referred to as somDispatchX. The syntax for these methods follows.

somDispatchX

```
somDispatchX(
             SOMAny *object,
             somId methodId,
             somId descriptorId,
             ...);
```

Parameters:

- ❑ object (SOMAny *) input—The object on which to invoke the method. (When used with user-supplied dispatch functions, the dispatch function may determine a different course of action, depending on the object and its class.)

- ❑ methodId (somId) input—The SOM ID corresponding to the method name. This can be obtained by calling SOM_IdFromString with the name of the method.

- ❑ descriptorId (somId) input—The SOM ID created from a descriptor (a string that describes the arguments and their types) associated with the method. When you are using the SOM-supplied dispatching algorithm, you can set it to NULL.

- ... —The remaining arguments, if any, that are needed by the method. Do not include the object pointer itself. Since the first argument to a method is always the object pointer, the dispatch function will handle that for you.

Figure 19-15 shows how the original example program, Client1.c (from Figure 19-3) would be implemented using somDispatchX.

```
/****************************************************************/
 *  CLIENT5.C -- 4 String class client program                  */
/****************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include "strng.h"

/*
 * Generic object state display routine.
 */
void dispString(char *s, int l)
{
   char *sp = "";

   if (s != NULL) {
      sp = s;
   }
   somPrintf("\nThe string is '%s'\n", sp);
   somPrintf("The string length is %d characters\n\n", l);
}

/*
 * MAINLINE
 */
main (long argc, char **argv)
{
   int     rc;
   int     ln;
   char   *pstr;
   SOMAny *stringObj;

   StringNewClass(0,0);       // Instantiate the class object
   stringObj = StringNew();   // Instantiate a string object

         // Set the value of the object - invoke _setString
   rc   = _somDispatchL(stringObj,
                SOM_IdFromString("setString"),
                NULL,
                "This is a string");
         // Get the string's value - invoke _getString
   pstr = _somDispatchA(stringObj,
                SOM_IdFromString("getString"),
                NULL);
      // Get the string's length - invoke _getStringLen
   ln   = _somDispatchL(stringObj,
                SOM_IdFromString("getStringLen"),
                NULL);

   dispString(pstr, ln);      // Display the string's data
   SOMFree(pstr);             // Free the intermediate buffer


      // Clear the string - invoke _clearString
   _somDispatchV(stringObj,
                 SOM_IdFromString("clearString"),
                 NULL);
      // Get the string's value - invoke _getString
   pstr = _somDispatchA(stringObj,
                SOM_IdFromString("getString"),
                NULL);

      // Get the string's length - invoke _getStringLen
   ln   = _somDispatchL(stringObj,
                SOM_IdFromString("getStringLen"),
                NULL);

   dispString(pstr, ln);       // Display the string's data

      // Set the string's value - invoke _setString
   rc   = _somDispatchL(stringObj,
                   SOM_IdFromString("setString"),
                   NULL,
                   "Here is a new string");

      // Get the string's value - invoke _getString
   pstr = _somDispatchA(stringObj,
                SOM_IdFromString("getString"),
                NULL);

      // Get the string's length - invoke _getStringLen
   ln   = _somDispatchL(stringObj,
                SOM_IdFromString("getStringLen"),
                NULL);

   dispString(pstr, ln);     // Display the string's data
   SOMFree(pstr);            // Free the intermediate buffer

   _somFree(stringObj);      // Free the String object
   exit(0);
}
```

Figure 19-15. *Client5.c (implementing Client.c with somDispatchX).*

If the number of arguments to be passed to the method is not known until runtime, you can assemble them into a variable-length argument list that can be passed to somDispatchX.

If you are coding in C, you can use the C feature that permits a variable number of parameters by listing all the arguments and letting C build the actual structure. For example, if you wanted to invoke a method called TwoParmMethod that called for two parameters and returned nothing, you could code the following:

```
_somDispatchV(anObj,                              // object ptr
       SOM_IdFromString("TwoParmMethod"),         // method name
       NULL,                                      // descriptor
       1,                                         // argument 1
       10);                                       // argument 2
```

This dynamic dispatch facility is somewhat restricted by the fact that you need to know at least what type of value is returned by the method in order to use it.

This restriction prevents you from writing a dispatching routine that is completely dynamic. (Not only is the return type restricted, but you must pass an indicator of that type at runtime in order to know which somDispatch method to call.) SOM, of course, lets you call a method that returns any type, but that type must be known at compile time.

DLLs

Class implementations are usually packaged as DLLs rather than EXEs. SOM provides several functions for making it easy to use classes from DLLs. One problem with packaging classes in DLLs is that SOM deals with class names dynamically, not statically. This means that when you invoke a method, SOM must know the name of the DLL in which the class implementation code resides.

If the client program is linked with the class implementation's import library, the DLL is loaded at client program load time. However, if you develop a dynamic SOM client, one that does not explicitly reference a class's bindings during compile time, SOM cannot load-time link with the DLL using the DLL's import library. In this case, SOM has two ways to find the DLL containing the class implementation.

One way is to require the file-stem portion of the DLL name to be the same as the class name (the file-stem is defined in the CLASS section of the OIDL file). For example, if the class were named Foo the DLL would be named FOO.DLL. To find the class implementation for the FOO class, SOM would search the directories specified in the LIBPATH OS/2 environment variable for a file named FOO.DLL.

However, this will not work if you want to package several class implementations into one DLL. The second way lets you explicitly load the DLL. SOM provides several methods and functions for handling DLLs, and you would use them to find and load the DLL for SOM.

Several methods will be described here, but you typically would need to use only SOMLoadModule, SOMDeleteModule, and somClassInitFuncName.

SOMLoadModule, SOMDeleteModule, and somClassInitFuncName are replaceable. The available methods and functions are summarized in Table 19-9.

`somFindClass`
`somFindClsInFile`
`SOMLoadModule`
`SOMDeleteModule`
`SOMLoadClassFile`
`somClassInitFuncName`

Table 19-9. *Methods and functions for DLL handling.*

SOMFindClass was described earlier in this chapter in *Class Hierarchy Methods*. Recall that it will load the DLL containing the class if the DLL is not loaded. Using SOMFindClass requires that the DLL have the same file-stem name as the class sought.

SOMFindClsInFile goes a step further by accepting as a parameter the name of the DLL file in which the class implementation is expected to reside. This method can be used to load a DLL containing more than one class implementation. The syntax for somFindClsInFile follows.

somFindClsInFile

```
        SOMClass * somFindClsInFile(
                                    SOMClassMgr * classMgrObj,
                                    somID classID,
                                    int majorversion,
                                    int minorversion,
                                    char * file)
```

Parameters:

- ❑ clsMgrObject (SOMClassMgr *) input—A pointer to an object of the SOMClass class or its descendants. While any class object for a metaclass is a descendant of the SOMClass class, you'll typically use somClassMgrObject, which is one of the global objects instantiated during SOM initialization. Client3.C in Figure 19-11 shows how to access it in C using the extern keyword.
- ❑ classID (somID) input—A SOM ID created from the class name to be found; must be the internal representation of the class name, not the representation in the form of a pointer to a string.
- ❑ majorversion (int) input—The class implementation major version number or 0. If it is not 0, the class implementation that is located must have the same major version number. This is specified in the class's CSC file.
- ❑ minorversion (int) input—The class implementation minor version number or 0. If it is not 0, the class implementation that is located must have the same minor version number. This is specified in the class's CSC file.
- ❑ file (char *) input—A null terminated string containing a DLL file name. Under OS/2, the DLL will not be found unless the file resides in one of the directories in the LIBPATH environment variable.

Whenever SOM loads a DLL, it calls a class initialization function. The default name of the class initialization function is SOMInitModule. When you put several classes into one DLL, you must then code your own class initialization function, named SOMInitModule, that calls <class>NewClass for each class in the DLL. Any other DLL initialization needed by your DLL should also reside in SOMInitModule. (Use

SOMClassInitFuncName, described next, to change the name from the default, SOMInitModule, to one of your choosing.)

As an example, suppose the String class and the calculator class (from Chapter 18) were put into one DLL. Figure 19-16 shows what the SOMInitModule for the DLL might look like. Notice the different parameters for the major and minor version numbers. Specifying majorversion and minorversion means you require a particular level of that class implemention. Specifying 0 means you don't care. (You specify the major and minor version numbers for a class in the CLASS section of the OIDL file.)

```
#include <strng.h>
#include <calc.h>
#pragma linkage (SOMInitModule, system)
void SOMInitModule(int majorversion, int minorversion)
{
   StringNewClass(majorversion, minorversion);
   calculatorNewClass(0, 0);
}
```

Figure 19-16. *SOMInitModule.*

Whenever a DLL is loaded, some initialization should occur for the classes within the DLL. This can be as simple as instantiating class objects or as complex as a regular client program.

The initialization code is placed in a function in the DLL. SOMClassInitFuncName returns the name of this class initialization function. SOMInitModule is the default name.

If you want to use another name for the initialization function, you must replace SOMClassInitFuncName with a function that returns the other name you specify. (SOMClassInitFuncName has nothing to do with the classinit option in the OIDL CLASS section.) The syntax for SOMClassInitFuncName follows.

SOMClassInitFuncName

```
          char * SOMClassInitFuncName(
                                     void)
```

SOMLoadModule loads a particular DLL module and passes control to a function of your choosing if the load is successful. The function returns 0 if the call was successful and a system specific non-0 return code if it was not.

For OS/2 2.1, look at DosLoadModule for the return codes.

When you use SOMLoadModule, you should specify SOMInitModule or the result of the SOMClassInitFuncName function as the function to be given control after the DLL is loaded. The syntax for SOMLoadModulefollows.

SOMLoadModule

```
int SOMLoadModule(
                        char * className,
                        char * fileName,
                        char * functionName,
                        int majorversion
                        int minorversion,
                        somToken * modhandle)
```

Parameters:

- ❑ className (char *) input—A null terminated string containing the class name.
- ❑ fileName (char *) input—A null terminated string containing the DLL file name.
- ❑ functionName (char *) input—A null terminated string containing the name of the function that is to gain control when the DLL is loaded successfully.
- ❑ majorversion (int) input—The class implementation major version number from the CSC file, or 0.
- ❑ minorversion (int) input—The class implementation minor version number from the CSC file, or 0.
- ❑ modHandle (somToken) output—Pointer to a variable that will receive the module handle of the loaded DLL.

SOMDeleteModule frees a previously loaded DLL. The function returns 0 if the call was successful and a system specific non-0 return code if it was not. For OS/2 2.1, look at DosFreeModule for the return codes. The syntax for SOMDeleteModule follows.

SOMDeleteModule

```
          int SOMDeleteModule(
                                    somToken modHandle)
```

Parameter:

- ❑ modHandle (somToken) input—The module handle of the DLL to be deleted. This handle was returned from SOMLoadModule.

SUMMARY

This chapter covers many of the methods, macros, and functions provided by SOM. You should be able to do quite a bit with SOM now. The online help facility for programmers that comes with SOM lists and covers all of the interfaces for SOM. If you want to do something that isn't covered here, look online or in the OS/2 2.0 Technical Library for the reference manual on SOM.

The interfaces covered in this chapter are the following:

NAME	TYPE
<class>MethodDebug	Macro
parent_<methodname>	Macro
SOM_Assert	Macro
SOM_AssertLevel	Global Variable
SOMCalloc	Function
somCheckID	Macro
SOMClassInitFuncName	Function
SOM_CompareIds	Macro
somDescendedFrom	Method
SOMDeleteModule	Function
somDispatchA	Method
somDispatchD	Method
somDispathL	Method
somDispathV	Method
somDumpSelf	Method
somDumpSelfInt	Method
SOMError	Function
SOM_Error	Macro
SOM_Expect	Macro

NAME	TYPE
somFindClass	Method
somFindClsInFile	Method
somFree	Method
SOMFree	Function
somGetClass	Method
somGetClassName	Method
somGetParent	Method
SOM_IdFromString	Macro
somInit	Method
somIsA	Method
somIsInstanceOf	Method
SOMLoadClassFile	Function
SOMLoadModule	Function
SOMMalloc	Function
somNew	Method
SOMOutCharRoutine	Function
somPrintf	Function
SOMRealloc	Function
somSelf	Local Variable
SOM_StringFromID	Macro
SOM_Test	Macro
SOM_TestC	Macro
somThis	Local Variable
SOM_TraceLevel	Global Variable
somUninit	Method
SOM_WarnLevel	Global Variable
SOM_WarnMsg	Macro

CHAPTER 20

Distributed Computing Environment (DCE)

"But be not afraid of greatness: some are born great, some achieve greatness and some have greatness thrust upon 'em." —*Shakespeare*

INTRODUCTION

Distributed Computing Environment (DCE) is a software product commissioned by the Open Software Foundation (OSF) in Cambridge, MA. Various computer companies provide elements of DCE to the OSF, which selects and integrates the most successful submissions. Companies can then license all or part of DCE and package it with their own software products. The OSF also publishes extensive documentation for DCE through Prentice Hall so that interested parties can read about it before making an investment.

The purpose of DCE is to supply a software bridge that minimizes the differences between non-compatible platforms. Using DCE as a kind of middleman, one machine can talk to another of a completely different type.

More importantly, processing and storage of data can take place on different machines. The end user never has to know the gory details.

Hosts, mid-range systems, workstations, and personal computers running different operating systems can interact by means of DCE. These heterogeneous machines keep their own operating systems and networking communications intact.

You can use DCE for such functions as sharing data, distributing the processing of data across different machines, and ensuring that data is secure. An application running with DCE can be distributed over several machines on the network, and only the DCE administrator knows where and how.

This chapter provides a brief introduction to the following topics:

❑ The architectural and conceptual view of DCE

❑ The components of DCE

❑ The programmer's view of DCE

ARCHITECTURAL AND CONCEPTUAL VIEW

Architecturally, DCE is the layer between a distributed application at the top and the operating system with its transport mechanism at the bottom of the layers. While many operating systems contain a built-in transport system, others, such as OS/2, do not. Before you can install and use DCE, you must supply and configure the networking and transport software that connects machines running OS/2. Currently, IBM recommends the Multi-Protocol Transport Service (MPTS) for use with DCE on OS/2. The DCE layer then provides the services that allow a distributed application to work on unlike systems.

Conceptually, DCE is based on three main ideas:

❑ A client-server relationship among various heterogeneous platforms in a network, known as the *client-server concept*

❑ A method of making remote procedure calls to unlike machines in a network, known as the *remote procedure call (RPC) concept*

❑ A secure means of locating, sharing, and protecting data across different systems in a network, known as the *data-sharing concept*

Client-Server Concept

DCE uses the well-established concept of a client machine making requests of a server machine, which responds to those requests. The client and server roles are not tied to specific machines. At one time, Machine A can function as a client, for example in requesting a copy of a file from Machine B, which acts as a server. At another time, Machine A can function as a server, for example in printing a file for its client, Machine B. However, it is customary to set up certain machines as specific types of servers that provide specific types of services, such as the following:

- ❑ A file server, which stores large numbers of files that are accessible to clients on the network
- ❑ A print server, which is connected to printers that can be used by clients on the network
- ❑ A security server, which verifies the identity of potential users of the network
- ❑ A directory server, which stores and locates names and addresses of machines on the network
- ❑ A time server, which provides the correct time that machines on the network can use to synchronize their clocks

It is also possible to install more than one server on a particular machine. Typically, a distributed computing environment has far more clients than servers. A server is usually configured as a continuous process that constantly listens for clients, while a client runs discontinuously.

Client machines are usually set up as several types of client so that they can make requests of all the various servers. Clients do not run continuously like servers because users need access to their machines while the distributed computing is taking place. A client is typically configured as a library of routines containing calls to various types of servers.

RPC Concept

The remote procedure call (RPC) resembles a call to a subroutine from the main program in that the main program is interested only in the results it receives from the subroutine, not where the subroutine resides or how it achieves its results.

When a distributed application makes an RPC, the call takes on a form in which it can travel over a network, make a request of a server, wait for the results, and carry these results back over the network to the caller.

RPC is the basic communications mechanism of DCE. It provides for the processing of a DCE application to be distributed over more than one machine in a network. It is discussed in more detail later in this chapter in *DCE RPC*.

Data Sharing Concept

In data sharing, the data is distributed across more than one machine, regardless of where the data is to be processed. A copy of data residing on a server, for example, can be sent, by means of RPC, to the client that needs it. The client can then access the data as if it were local.

Multiple copies of data and multiple access to the data can take place simultaneously. However, DCE provides a means of maintaining consistency among the copies as well as managing conflicting requests for access.

Two components of DCE depend on the data sharing concept: the Directory Service, which has already been implemented for OS/2, and the Distributed File Service, which has not been implemented for OS/2.

THE COMPONENTS OF DCE

The components of DCE intertwine and depend on each other to such an extent that it is difficult, if necessary, to separate them for the purpose of this discussion. Figure 20-1 shows how these components interact.

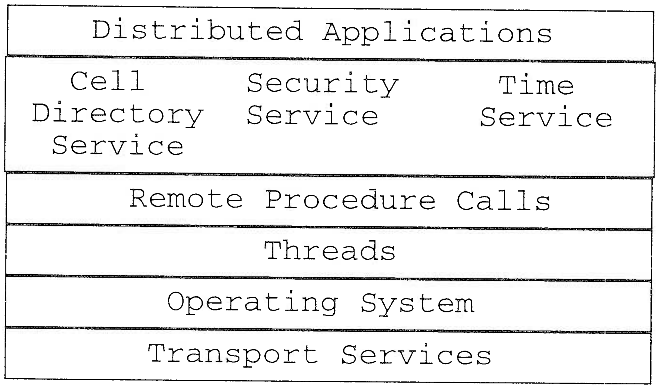


Figure 20-1. Components of DCE.

The top layer, Distributed Applications, and the bottom two layers, Operating System and Transport Services, are not part of DCE. However, they interact with Threads, Remote Procedure Call, and the three services (Cell Directory, Security, and Time).

The components that have been implemented for OS/2 are the following:

- ❑ DCE RPC
- ❑ DCE Directory Service
- ❑ DCE Security Service
- ❑ DCE Distributed Time Service

DCE RPC

For an application to run in the distributed computing environment, code on the client side and code on the server side, known as *the client stub* and *the server stub*, must communicate with each other across the network. RPC provides a communications or calling interface that spares you the effort of dealing with the differences between unlike hardware architectures, operating systems, computing languages, and communications protocols. RPC handles such details as splitting and rejoining calls, locating the correct receiver of the call, and providing for integrity of the information transmitted in the call. These built-in facilities enable you to write code that does not have to be rewritten for different types of machines in a heterogeneous environment.

RPC consists of the following:

- ❑ The Interface Definition Language (IDL) and the IDL Compiler
- ❑ The RPC Runtime Library
- ❑ The Name Service Independent Application Programming Interface (NSI API)
- ❑ The RPC daemon (a *daemon* is a continuously running process)
- ❑ The RPC control program
- ❑ The Universal Unique Identifier (UUID) facility
- ❑ Authenticated RPC

IDL and the IDL Compiler

IDL is a high-level language used for setting up an IDL file, which contains a server's interfaces. These interfaces define the set of operations that the server can perform for a particular distributed application. The IDL file also defines supported data types and constant declarations. The IDL Compiler makes the interfaces usable either as object code or as C language code.

Output from the IDL Compiler is split between the client side and the server side. It consists of a client stub (client code module), a server stub (server code module), and a header file.

The client stub is used by any client application you write that makes calls to the operations in a specific IDL file. The client stub and the RPC Runtime Library translate a call into a form that can travel across the network and make a request of the correct server. The server stub is used by code that implements the server's operations as specified in an IDL file.

Typically, you will write an implementation, or application routine, for each operation. When the server receives a request from a client on the network, the server stub translates the request and its arguments into a form in which it can make a call to the server application routine.

The RPC Runtime Library

The RPC Runtime Library contains a set of operations that provide the means to access a name service, to obtain security information, and to manage servers. All RPC applications make use of this library.

The Name Service Independent API

The NSI API provides APIs for use with a name service, such as the Cell Directory Service (CDS). You can create an entry for a server, for example, that contains the server's network address, its endpoint (the address on a host of a server instance), its RPC protocol sequence, and its transfer syntax (a set of encoding and decoding rules for transmitting data across a network).

The RPC Daemon

The RPC daemon (**rpcd**) is a continuously running server process that looks up the endpoint of a local server. This feature saves storage overhead and makes it unnecessary for you to know the endpoint before you make an RPC request.

The RPC Control Program

The RPC Control Program (**rpccp**) is a utility that administers the RPC daemon. The RPCCP provides the means to add, update, delete, and browse RPC attributes of entries stored in the CDS *namespace*, which is discussed later in this chapter in *DCE Directory Service*. RPC attributes include:

- ❑ NSI binding attributes (information about the location of a server)
- ❑ NSI group attributes (information about a set of servers that form an RPC group)
- ❑ NSI object attributes (information about objects such as databases, directories, or devices)
- ❑ NSI profile attributes (information about a set of RPC profile elements in a name service entry)

UUID Facilities

UUID Facilities create universal unique identifiers for such RPC elements as interfaces and objects. These identifiers are stable, permanent names that do not change across networks or over time. The **uuidgen** program is used to create UUIDs.

Authenticated RPC

Authenticated RPC, a part of the DCE Security Service, provides for security, data privacy, and data integrity during a remote procedure call. Authentication refers to a method of verifying the identity of a client that makes a request of a server. An authenticated RPC client, one that has the correct authorization, can access various levels of services on a server. For more on authentication, see the discussion later in this chapter on *DCE Security Service*.

DCE Directory Service

A directory service in a distributed computing environment acts as a centralized storehouse of information about such resources as users, computers, and services. A directory contains an entry for each resource, which consists of the name of the entry and such attributes as location, type, and operation.

These entries are stored in a hierarchical order in a database called a *clearinghouse*, which is located in the *namespace* that is shared with security and time servers. This feature enables an application program to look up the location of a server, for example, instead of requiring you to hardcode the address and then change the code if the location changes.

Two important features of the DCE Directory Service are *distribution* and *replication*. Information about resources can be distributed across more than one machine, which means that you can store some types or some parts of the namespace in one place and other types or parts in another place. This feature lets you store a certain type of directory information where it is most likely to be needed. Replication means that you can store multiple copies of namespace information on different machines. This feature ensures that the information is always available, even if one machine goes down.

The DCE Directory Service also takes into account your need to have consistent directory information available at all times. The distributed, replicated information is kept in the same state as the original information, which can change frequently in a large network. Figure 20-2 shows how the client and the server use the Directory Service. An explanation of the numbers follows.

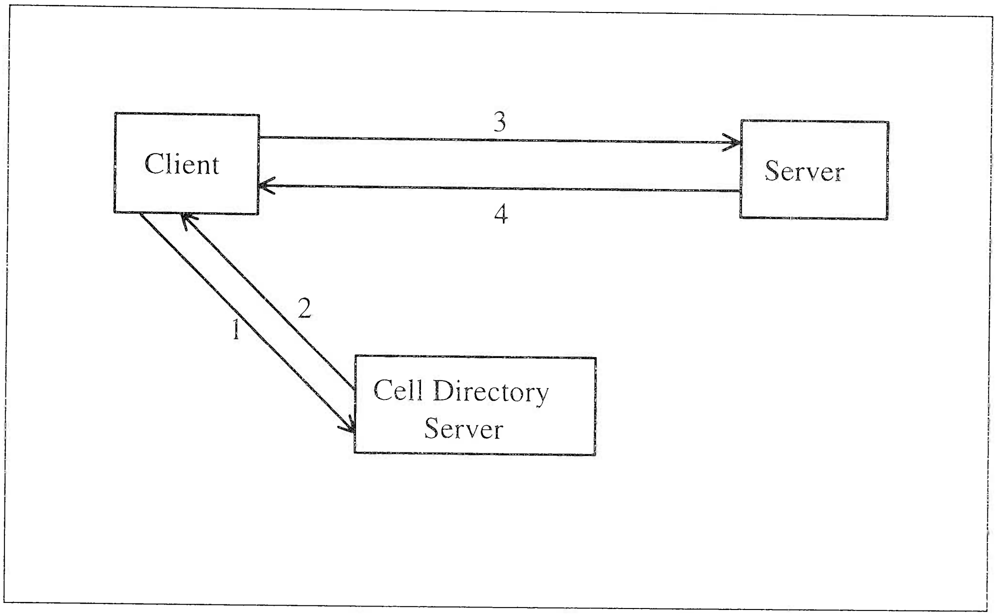


Figure 20-2. *The Client uses the Cell Directory Service.*

1. The Client requests the name and location of the Server.
2. The Cell Directory Server responds.
3. The Client, using this response, requests some service from the Server.
4. The Server returns the completed request for the service to the Client.

The DCE Directory Service is implemented for OS/2 as the Cell Directory Service (CDS). A cell is a group of users, machines, and resources that have something in common and some reason for being grouped together. Examples are cells set up for the local Accounting Department or for company-wide Quality Control personnel. Since a cell is an administrative entity, its members should have similar purposes and should require similar DCE services.

CDS consists of the CDS server, which maintains the information about object names for its cell, the CDS clerk, which handles requests from client applications, and several programs that perform administrative tasks. Any requests to access, create, and change name information are handled by the following CDS daemons:

- ❑ **cdsd**, the CDS server daemon, runs continuously on the CDS server, which stores the database of directory information.

- ❑ **cdsclerk**, the CDS clerk daemon, runs on the CDS client and acts as a middleman between a client application and the CDS server.

- ❑ **cdsadv,** the CDS advertiser daemon on the client machine, provides the interface between a client application that makes a request and the **cdsd** daemon on the server, which responds to the request.

- ❑ **cdscp,** the CDS Control Program, provides the means for DCE administrators to access and control server.

DCE Security Service

The DCE Security Service ensures that communications between clients and servers are secure and that access to resources in the distributed system is granted only to requests from authorized users.

A distributed network may look wide open and vulnerable to interlopers, but the DCE Security System provides several means of controlling access to system resources. Figure 20-3 shows some of these controls.

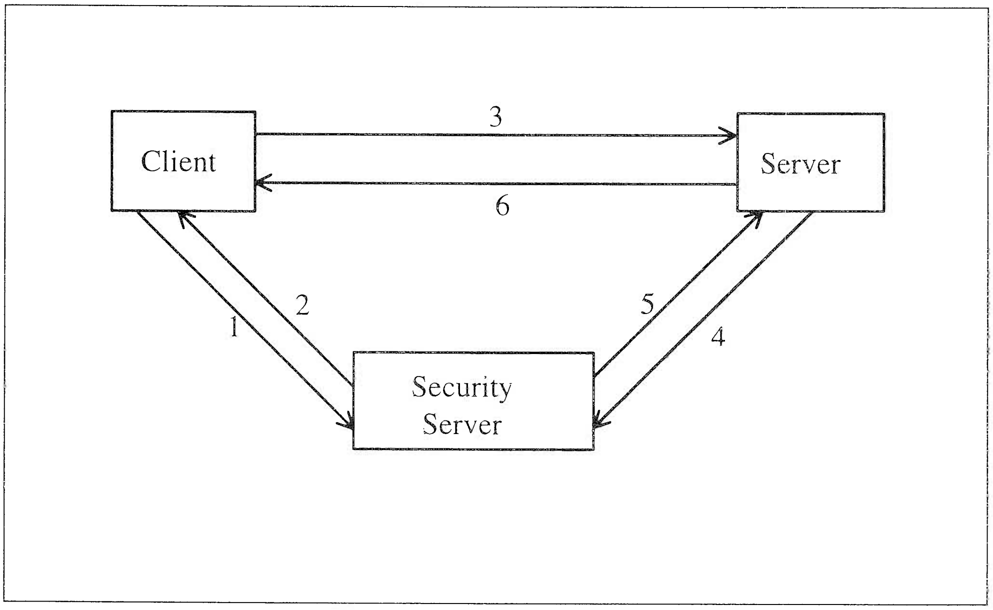


Figure 20-3. *The Client and Server use the Security Service.*

1. The Client requests authentication from the Server.
2. The Security Server responds with authentication.
3. The Client, using this response, requests some service from the Server.
4. The Server requests verification of the Client's identity.
5. The Security Service responds with confirmation.
6. The Server returns the completed request for the service to the Client.

In addition, the DCE Security Service guarantees the integrity and the privacy of communications between different machines by doing one or more of the following:

❑ The Security Service *authenticates*, or confirms tht identity of, two processes running on different machines so that they can trust each other. This identification service is provided by the Authentication Service.

❑ The Security Service verifies exactly which level of services an authenticated client application is allowed to access. The Privilege Service, along with the Login Facility, discussed below, and Authenticated RPC, authorizes certain clients for certain services.

- ❑ The Security Service lets DCE administrators access, update, add, and delete user entries in the Security database by means of the Registry Service.
- ❑ The Security Service maintains lists of users who are authorized for access to certain resources in Access Control Lists (ACLs).
- ❑ The Security Service provides a Login Facility based on a user's password and security credentials.

These, along with the security server daemon, **secd**, and the security client daemon, **sclientd**, ensure that the network is secure.

DCE Distributed Time Service

The DCE Distributed Time Service (DTS) provides a means for synchronizing the clocks on all the machines in the network. Each cell must have one DTS server, but at least three are required in order to determine if one of the three is defective. DTS consists of time servers, time clerks, and an API for managing access to time servers.

THE PROGRAMMER'S VIEW OF DCE

From a programmer's perspective, DCE offers a useful application development environment. Before DCE was available, you had to do all the work when you were writing an application that spanned more than one machine.

On the client side, for example, you had to know whether the services you needed were local or remote. You had to locate each service you needed and select a protocol by which to access that service.

Then you had to send the necessary parameters to the service, make the connection to the service, and handle all the details of security. You had to translate or encode your data, handle communications errors, make the request for service, and handle any returns.

For the communications subsystem, you had to handle network routing and data transport. Then on the server side, you had to take care of communications errors, translate the data sent, handle security, pull out the request for service, provide the service, and handle the return to the client application.

By using DCE, however, most of these chores are done for you. Your client application has to do two things: Make the call to the needed service, and do something with what is returned. All the other tasks are handled by DCE RPC, DCE Directory Services, and DCE Security Services.

The administrator of a DCE network may have tasks similar to those for any other kind of network, but the application programmer gets a real break in using DCE.

INDEX

VNR COMPUTER LIBRARY

OS/2/Programming

Find out the the latest on:
- CID (Configuration, Installation, and Distribution)
- SOM (System Object Model)
- Control Program API calls (Kernel Calls)
- Memory Management
- Multitasking
- Interprocess Communication (IPC)
- Distributed Computing Environment

Covers both standard and advanced programming topics for OS/2!

Writing applications using OS/2 2.1 is made easy with this programmer's guide. It's packed with parameters, flags, data structures, code examples, and sample output that enable you to write programs.

Coverage ranges from standard programming topics such as files, threads, semaphores, pipes, and porting to the latest advancements in binding, customization, compiling, and language independence. Sections on building programs for LAN and object-oriented environments address how to use CID to redirect installation of OS/2 applications and how to use the SOM to implement object-oriented programming. You'll also become familiar with the architecture and functionality of the OS/2 2.1 Control Program and the implementation of Distributed Computing Environment on OS/2.

All you need is some experience coding in a high-level language such as C to put this guide to work. It's a valuable reference for all beginning and intermediate programmers who program in OS/2 2.1.

About the Authors

Jody Kelly, Craig Swearingen, Dawn Bezviner, and Theodore Shrader are programmers and technical writers at IBM Corporation.

Tap the full power of OS/2 by learning how to:
- Port existing 16-bit applications to OS/2 2.1
- Create dynamic link libraries and executable files
- Design class hierarchies and use the runtime environment
- Override, write, and compile CSC files
- Create, allocate, open, read, write, purge, and close queues
- Suspend threads, set intervals, and manage asynchronous timers
- Manage exceptions and messages
- Correct errors in code

Cover design and illustration by Jon Herder

VAN NOSTRAND REINHOLD
115 Fifth Avenue, New York, NY 10003

PUBLISHED BY JOHN WILEY AND SONS,
0471131164
00000
9 780471 131168